# 分布式缓存，Redis，MemCache

1. Redis中，并不是所有的数据都一直存储在内存中的，这是和Memcached相比一个最大的区别。

2. Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储。

3. Redis支持数据的备份和持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

Redis在很多方面具备数据库的特征，或者说就是一个数据库系统，而Memcached只是简单的K/V缓存

Redis是基于内存、可持久化的日志型、Key-Value[数据库](http://baike.baidu.com/view/1088.htm) 高性能存储系统，并提供多种语言的API.

**出现背景**

数据结构(Data Structure)需求越来越多, 但memcache中没有, 影响开发效率

性能需求, 随着读操作的量的上升需要解决，经历的过程有:   
数据库读写分离(M/S)–>数据库使用多个Slave–>增加Cache (memcache)–>转到Redis

解决写的问题：   
水平拆分，对表的拆分，将有的用户放在这个表，有的用户放在另外一个表；

可靠性需求   
Cache的"雪崩"问题让人纠结   
Cache面临着快速恢复的挑战

开发成本需求   
Cache和DB的一致性维护成本越来越高(先清理DB, 再清理缓存, 不行啊, 太慢了!)   
开发需要跟上不断涌入的产品需求   
硬件成本最贵的就是数据库层面的机器，基本上比前端的机器要贵几倍，主要是IO密集型，很耗硬件；

维护性复杂   
一致性维护成本越来越高；   
BerkeleyDB使用B树，会一直写新的，内部不会有文件重新组织；这样会导致文件越来越大；大的时候需要进行文件归档，归档的操作要定期做；   
这样，就需要有一定的down time；

# 本地线程

从中我们可以发现这个Map的key是ThreadLocal变量，value为用户的值，并不是网上大多数的列子key是线程的名字或者标识

到这里，我们就可以理解ThreadLocal究竟是如何工作的了

1.Thread类中有一个成员变量叫做ThreadLocalMap，它是一个Map，他的Key是ThreadLocal类

2.每个线程拥有自己的申明为ThreadLocal类型的变量,所以这个类的名字叫'ThreadLocal'：线程自己的（变量）

3.此变量生命周期是由该线程决定的，开始于第一次初始（get或者set方法）

4.由ThreadLocal的工作原理决定了：每个线程独自拥有一个变量，并非共享或者拷贝

无论如何，要编写一个多线程安全(Thread-safe)的程序是困难的，为了让线程共享资源，必须小心地对共享资源进行同步，同步带来一定的效能延迟，而另一方面，在处理同步的时候，又要注意对象的锁定与释放，避免产生死结，种种因素都使得编写多线程程序变得困难。

尝试从另一个角度来思考多线程共享资源的问题，既然共享资源这么困难，那么就干脆不要共享，何不为每个线程创造一个资源的复本。将每一个线程存取数据的行为加以隔离，实现的方法就是给予每个线程一个特定空间来保管该线程所独享的资源

什么是ThreadLocal？ www.2cto.com

顾名思义它是local variable（线程局部变量）。它的功用非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，是每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量。

使用场景

To keep state with a thread (user-id, transaction-id, logging-id)

To cache objects which you need frequently

ThreadLocal类

它主要由四个方法组成initialValue()，get()，set(T)，remove()，其中值得注意的是initialValue()，该方法是一个protected的方法，显然是为了子类重写而特意实现的。该方法返回当前线程在该线程局部变量的初始值，这个方法是一个延迟调用方法，在一个线程第1次调用get()或者set(Object)时才执行，并且仅执行1次。ThreadLocal中的确实实现直接返回一个null：

ThreadLocal的原理

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单，在ThreadLocal类中有一个Map，用于存储每一个线程的变量的副本。比如下面的示例实现：

public class ThreadLocal

{

　 private Map values = Collections.synchronizedMap(new HashMap());

　 public Object get()

　 {

　　 Thread curThread = Thread.currentThread();

　　 Object o = values.get(curThread);

　　 if (o == null && !values.containsKey(curThread))

　　 {

　　 　 o = initialValue();

　　 　 values.put(curThread, o);

　　 }

　　 return o;

　}

　public void set(Object newValue)

　{

　　 values.put(Thread.currentThread(), newValue);

　}

　public Object initialValue()

　{

　　 return null;

　}

}

ThreadLocal 的使用

使用方法一：

Hibernate的文档时看到了关于使ThreadLocal管理多线程访问的部分。具体代码如下

1. public static final ThreadLocal session = new ThreadLocal();

2. public static Session currentSession() {

3. Session s = (Session)session.get();

4. //open a new session,if this session has none

5. if(s == null){

6. s = sessionFactory.openSession();

7. session.set(s);

8. }

return s;

9. }

我们逐行分析

1。 初始化一个ThreadLocal对象，ThreadLocal有三个成员方法 get()、set()、initialvalue()。

如果不初始化initialvalue，则initialvalue返回null。

3。 session的get根据当前线程返回其对应的线程内部变量，也就是我们需要的net.sf.hibernate.Session（相当于对应每个数据库连接）.多线程情况下共享数据库链接是不安全的。ThreadLocal保证了每个线程都有自己的s（数据库连接）。

5。如果是该线程初次访问，自然，s（数据库连接）会是null，接着创建一个Session，具体就是行6。

6。创建一个数据库连接实例 s

7。保存该数据库连接s到ThreadLocal中。

8。如果当前线程已经访问过数据库了，则从session中get()就可以获取该线程上次获取过的连接实例。

使用方法二

当要给线程初始化一个特殊值时，需要自己实现ThreadLocal的子类并重写该方法，通常使用一个内部匿名类对ThreadLocal进行子类化，EasyDBO中创建jdbc连接上下文就是这样做的：

public class JDBCContext{

private static Logger logger = Logger.getLogger(JDBCContext.class);

private DataSource ds;

protected Connection connection;

private boolean isValid = true;

private static ThreadLocal jdbcContext;

private JDBCContext(DataSource ds){

this.ds = ds;

createConnection();

}

public static JDBCContext getJdbcContext(javax.sql.DataSource ds)

{

if(jdbcContext==null)jdbcContext=new JDBCContextThreadLocal(ds);

JDBCContext context = (JDBCContext) jdbcContext.get();

if (context == null) {

context = new JDBCContext(ds);

}

return context;

}

private static class JDBCContextThreadLocal extends ThreadLocal {

public javax.sql.DataSource ds;

public JDBCContextThreadLocal(javax.sql.DataSource ds)

{

this.ds=ds;

}

protected synchronized Object initialValue() {

return new JDBCContext(ds);

}

}

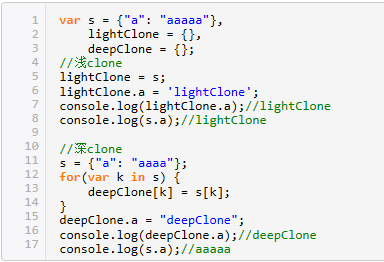
}

使用单例模式，不同的线程调用getJdbcContext()获得自己的jdbcContext，都是通过JDBCContextThreadLocal 内置子类来获得JDBCContext对象的线程局部变量，这个变

# JS重点：CSS,闭包，AMD，继承，原型链，深度克隆,this的用法,node.js

前端技术虽然在不断发展之中，却一直没有质的飞跃。除了已有的各大著名框架，比如Dojo，JQuery，ExtJs等等，很多公司也都有着自己的前端开发框架。这些框架的使用效率以及开发质量在很大程度上都取决于开发者对其的熟悉程度，以及对JavaScript的熟悉程度，这也是为什么很多公司的技术带头人都喜欢开发一个自己的框架。开发一个自己会用的框架并不难，但开发一个大家都喜欢的框架却很难。从一个框架迁移到一个新的框架，开发者很有可能还会按照原有框架的思维去思考和解决问题。这其中的一个重要原因就是JavaScript本身的灵活性：框架没办法绝对的约束你的行为，一件事情总可以用多种途径去实现，所以我们只能在方法学上去引导正确的实施方法。庆幸的是，在这个层面上的软件方法学研究，一直有人在去不断的尝试和改进，CommonJS就是其中的一个重要组织。他们提出了许多新的JavaScript架构方案和标准，希望能为前端开发提供银弹，提供统一的指引。

　　AMD规范就是其中比较著名一个，全称是Asynchronous Module Definition，即异步模块加载机制。从它的规范描述页面看，AMD很短也很简单，但它却完整描述了模块的定义，依赖关系，引用关系以及加载机制。从它被requireJS，NodeJs，Dojo，JQuery使用也可以看出它具有很大的价值，没错，JQuery近期也采用了AMD规范。在这篇文章中，我们就将介绍AMD的性质，用法，优势以及应用场景。从AMD中我们也能学习到如何在更高层面去设计自己的前端应用。





出于种种原因，我们有时候需要得到函数内的局部变量。但是，前面已经说过了，正常情况下，这是办不到的，只有通过变通方法才能实现。

那就是在函数的内部，再定义一个函数。函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，

f2内部的局部变量，对f1 就是不可见的。这就是Javascript语言特有的“链式作用域”结构（chain scope），子对象会一级一级地向上寻找所

有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。既然f2可以读取f1中的局部变量，那么只要把f2作为返回

值，我们不就可以在f1外部读取它的内部变量了吗

Javascript是一种web前端语言，主要用于[web开发](http://www.baidu.com/s?wd=web%E5%BC%80%E5%8F%91&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkmHw-rjD4P1ndrjF-nWfz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHT1n1n1rjm1rHmvrjbLPjDYPs)中，由浏览器解析执行  
  
[Node.js](http://www.baidu.com/s?wd=Node.js&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkmHw-rjD4P1ndrjF-nWfz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHT1n1n1rjm1rHmvrjbLPjDYPs)是一个可以快速构建网络服务及应用的平台，是用Javascript语言构建的服务平台，可用于后端建立服务器  
他们都用的是js语法，前者主要应用前端后者主要应用与后端

**Node.js**

1.优点

（1）高并发（最重要的优点）

（2）适合I/O密集型应用

2.缺点

（1）不适合CPU密集型应用；CPU密集型应用给Node带来的挑战主要是：由于JavaScript单线程的原因，如果有长时间运行的计算（比如大循环），

将会导致CPU时间片不能释放，使得后续I/O无法发起；

解决方案：分解大型运算任务为多个小任务，使得运算能够适时释放，不阻塞I/O调用的发起；

（2）只支持单核CPU，不能充分利用CPU

（3）可靠性低，一旦代码某个环节崩溃，整个系统都崩溃（原因：单进程，单线程）

解决方案：（1）Nnigx反向代理，负载均衡，开多个进程，绑定多个端口；开多个进程监听同一个端口，使用cluster模块；

（4）开源组件库质量参差不齐，更新快，向下不兼容

（5）Debug不方便，错误没有stack trace

# Linux

ls　 显示文件或目录

ls -l 列出文件详细信息l(list)

ls -a 列出当前目录下所有文件及目录，包括隐藏的a(all)

mkdir test1; 简单创建

mkdir -m 777 test2; 授予777权限

mkdir -v test3; 创建并且显示

mkdir -p -v aaa/bbb/ccc; 创建并且显示，父目录自动生成

rm demo.java 直接删除文件

rm -R -v aaa 递归删除，删除完成显示

rmdir -v czj 删除空目录，删除完成显示

rmdir -v -p aaa/bbb/ccc/ 递归删除空目录，注意删除的顺序，由子目录到父目录

cd /opt/huawei/hnas\_uoms/uoms/bin 进入目录

cd .. 上级目录

cd ../.. 上两级目录

touch -m hs\_err\_pid2874.log 只更新修改时间

touch demo.c 创建文件

echo 666 输出666换行

echo -n 666 输出666不换行

cat uomsDeploy.sh 查看整个文件内容

cat -n textfile1 > textfile2 把 textfile1 的档案内容加上行号后输入 textfile2 这个档案里

cp file1 file2 将文件file1复制成文件file2

cp -f file1 file2 强制复制的模式

cp -R file1 file2 将目录dir1复制成目录dir2

mv test.log test1.txt 将文件test.log改名为test1.txt

mv test1.txt test3 将文件test1.txt移到test3

mv -f log3.txt log2.txt 将文件file1改名为file2，即使file2存在，也是直接覆盖掉

mv -i log3.txt log2.txt 将文件file1改名为file2，file2存在，询问

mv dir1 dir2 目录的移动

find ./ -size -10G 文件size小于10G的文件或目录

find ./ -size +10K 文件size大于10M的文件或目录

find ./ -nogroup 查找文件的组ID不存在的文件

find ./ -group wade 查找组名为wade的文件或目录

find ./ -user czj 所有者为czj的文件或目录

find ./ -perm -o=r 其它用户权限有读权限的目录或文件

find ./ -perm -g=w 用户组权限有写权限的目录或文件

find ./ -perm -u=e 用户权限有执行权限的目录或文件

ind ./ -perm 664 权限为644的文件或目录

find ./ -empty -type f -print -delete 查找空文件并删除

find ./ -mmin -2 查找文件更新日时在距现在时刻二分以内的文件

find ./ -name \\*.zip 查找文件名匹配\*.zip的文件

find ./ -name test 查找文件名为test的文件

wc testfile testfile\_1 testfile\_2 #统计三个文件的信息 行数、字数、字符数

grep ‘test’ d\* 显示所有以d开头的文件中包含 test的行

grep ‘[a-z]\{5\}’ aa 显示aa所有包含每个字符串至少有5个连续小写字符的字符串的行

pwd 显示当前目录

clear 清除命令

more uomsDeploy.sh 分页显示uomsDeploy.sh；enter向下一行；Ctrl+f向下滚动一屏；Ctrl+b返回上一屏

less uomsDeploy.sh 分页显示uomsDeploy.sh；enter向下一行；空格向下滚动一屏；g第一行；G最后一行。

head -5 uomsDeploy.sh 显示开始5行

tail -10 /etc/passwd 读取最后10行

tail -f /var/log/messages 读取最后几行，不断读取新的内容，实时监控的作用

ctrl+c 退出

全文行号高亮： grep timerLockMapper -n --color run.log

监控最后行号高亮（只显示）：tail -f -n 1000 run.log |grep timerLockMapper -n --color

监控最后行号高亮（全部显示）：tail -f run.log | perl -pe 's/(request)/\e[1;31m$1\e[0m/g'

（2）系统管理

who 显示在线登陆用户

whoami 显示当前操作用户

hostname 显示主机名

uname 显示系统信息

top 动态显示当前耗费资源最多进程信息

ps 显示瞬间进程状态 ps -aux

ping www.baidu.com 测试网络连通

netstat 显示网络状态信息

clear 清屏

kill -9 pid 强制终止,杀死进程，可以先用ps 或 top命令查看进程的id，然后再用kill命令杀死进程。

shutdown -r 关机重启

shutdown -h 关机不重启

shutdown now 立刻关机

halt 关机

reboot 重启

（3）打包压缩

tar: 打包压缩

-c 归档文件

-x 压缩文件

-z gzip压缩文件

-j bzip2压缩文件

-v 显示压缩或解压缩过程 v(view)

-f 使用档名

tar -cvf /home/abc.tar /home/abc 只打包，不压缩

tar -zcvf /home/abc.tar.gz /home/abc 打包，并用gzip压缩

tar -jcvf /home/abc.tar.bz2 /home/abc 打包，并用bzip2压缩

tar –cvf mysql-5.0.tar mysql-----把整个目录mysql中文件打包到mysql-5.0.tar

tar – rf mysql-5.0.tar mysql.conf ------将文件mysql.conf增加到包mysql-5.0.tar

tar –uf mysql-5.0.tar mysql.conf --------用文件mysql.conf更新包中的文件mysql.conf

tar –xvf mysql-5.0.tar -------打开包mysql-5.0.tar中的文件到当前目录

（4）vim三种模式：命令模式(Esc)、插入模式(i)、底行模式 (ctrl+c 或者 :)

命令模式下：

:q 退出

:q! 强制退出

:wq 保存并退出

（5）用户及权限

/etc/passwd 存储用户账号

/etc/group 存储组账号

/etc/shadow 存储用户账号的密码

/etc/gshadow 存储用户组账号的密码

useradd 用户名

userdel 用户名

adduser 用户名

groupadd 组名

groupdel 组名

passwd root 给root设置密码

su root

su - root

/etc/profile 系统环境变量

bash\_profile 用户环境变量

.bashrc 用户环境变量

su user 切换用户，加载配置文件.bashrc

su - user 切换用户，加载配置文件/etc/profile ，加载bash\_profile

chgrp : 改变档案所属群组

chown : 改变档案拥有者; 将install.log的拥有者改为bin这个账号： chown bin install.log

chmod : 改变档案的权限, SUID, SGID, SBIT等等的特性

Linux档案的基本权限就有九个，分别是owner/group/others三种身份各有自己的read/write/execute权限

举例：档案的权限字符为 -rwxrwxrwx 这九个权限是三个三个一组的！其中，我们可以使用数字来代表各个权限，各权限的分数对照表如下：

r:4 　　w:2　　　x:1

每种身份(owner/group/others)各自的三个权限(r/w/x)分数是需要累加的，例如当权限为： [-rwxrwx---] 分数则是：

owner = rwx = 4+2+1 = 7

group = rwx = 4+2+1 = 7

others= --- = 0+0+0 = 0

所以我们设定权限的变更时，该档案的权限数字就是770啦！变更权限的指令chmod的语法是这样的：

u User，即文件或目录的拥有者；

g Group，即文件或目录的所属群组；

o Other，除了文件或目录拥有者或所属群组之外，其他用户皆属于这个范围；

a All，即全部的用户，包含拥有者，所属群组以及其他用户；

r 读取权限，数字代号为“4”;

w 写入权限，数字代号为“2”；

x 执行或切换权限，数字代号为“1”；

- 不具任何权限，数字代号为“0”；

chmod [who] [+ | - | =] [mode] 文件名

chmod 777 filename

chmod -R 777 /home/mypackage

$ chmod u+x file 给file的属主增加执行权限

$ chmod 751 file 给file的属主分配读、写、执行(7)的权限，给file的所在组分配读、执行(5)的权限，给其他用户分配执行(1)的权限

$ chmod u=rwx,g=rx,o=x file 上例的另一种形式

$ chmod =r file 为所有用户分配读权限

$ chmod 444 file 同上例

$ chmod a-wx,a+r file 同上例

$ chmod -R u+r directory 递归地给directory目录下所有文件和子目录的属主分配读的权限

$ chmod 4755 设置用ID，给属主分配读、写和执行权限，给组和其他用户分配读、执行的权限

# SQL调优

数据库系统的显著特点需要保存大量历史记录，系统内存在许多历史记录表，因此常常出现系统运行一段时间，表记录数达到一定数量后，系统响应明显变慢的现象。为避免这种情况的出现，在有完备的数据库对象设计下，还在考虑到系统稳定情况下数据的变化情况，针对这种变化，在编写SQL语句必须遵循一定的优化规则,并制定完备的数据管理机制。

2. 调优目的

利用Oracle的内部优化器工作机制，合理改进查询语句的组织方式，提高数据库系统的响应速度，实现前端画面及平台应用的快速响应能力；同时统一开发人员的数据库开发编码的统一、规范。

3. 调优原则

充分利用表索引，避免进行全表扫描；充分利用SGA共享池缓存机制，提前Oracle工作效率，充分利用结构化编程方式，提高查询的复用能力。

4. 调优方法

4.1 书写方式

4.1.1 相同功能、性能的SQL语句

ORACLE采用共享内存SGA的机制，因此ORACLE会对每个不同写法的SQL进行分析，并且占用共享内存;如果书写格式完全相同，则ORACLE只分析一次，遇到相同书写格式的SQL，会直接从共享内存中获取结果集;这样便能减少共享池的开销以及代码的复用。

处理方法：保证书写格式相同，包括大小写，空格位置，表别名等一致；将一些通用的SQL语句作为公共函数由其它函数调用的方式使用。

4.1.2 动态SQL

动态SQL采用变量动态绑定的方式，避免重复解析。

4.2 连接方式与顺序

4.2.1 表名顺序

多表查询时需要选择最有效率的表名顺序（基于规则的优化器有效），ORACLE的解析器按照从右到左的顺序处理FROM子句中的表名；因此写在最后的那张基础表最先被处理，即选择记录数最少的表作为基础表，首先,扫描FROM子句中最右的那个表，并对记录进行派序,然后扫描FROM子句中最后第二个表,最后将所有从第二个表中检索出的记录与第一个表中合适记录进行合并。

4.2.2 查询条件顺序

ORACLE采用自下而上的顺序解析WHERE子句,根据这个原理,表之间的连接必须写在其他WHERE条件之前, 那些可以过滤掉最大数量记录的条件适合写在WHERE子句的末尾。

4.3 语法和语义

4.3.1 SELECT子句中避免使用 ‘ \* ‘

使用’\*’ ,Oracle便会查询数据字典，依次解析各列名，应明确指定各列的名称，这样也便于理解。

4.3.2 使用表的别名(Alias)

多表查询时，使用表的别名，同样可以避免解析，避免歧义，提前效率。

4.4 函数和表达式

4.4.1 用>=替代>

如果DEPTNO上有一个索引,

高效:

SELECT \*

FROM EMP

WHERE DEPTNO >=4

低效:

SELECT \*

FROM EMP

WHERE DEPTNO >3

两者的区别在于, 前者DBMS将直接跳到第一个DEPT等于4的记录而后者将首先定位到DEPTNO=3的记录并且向前扫描到第一个DEPT大于3的记录.

4.4.2 使用DECODE函数来减少处理时间

使用DECODE函数可以避免重复扫描相同记录或重复连接相同的表.

4.4.3 用TRUNCATE替代DELETE

当删除表中的记录时,在通常情况下, 回滚段(rollback segments ) 用来存放可以被恢复的信息. 如果你没有COMMIT事务,ORACLE会将数据恢复到删除之前的状态(准确地说是恢复到执行删除命令之前的状况)

而当运用TRUNCATE时, 回滚段不再存放任何可被恢复的信息.当命令运行后,数据不能被恢复.因此很少的资源被调用,执行时间也会很短.

TRUNCATE只在删除全表或分区适用,TRUNCATE是DDL不是DML

4.5 常用关键字优先级

4.5.1 用EXISTS替代IN

用IN将启用全表扫描，使用EXISTS将利用索引，提高查询效率。

4.5.2 用NOT EXISTS替代NOT IN

用NOT IN将启用全表扫描，使用NOT EXISTS将利用索引，提高查询效率。

4.5.3 用表连接替换EXISTS

4.5.4 用EXISTS替换DISTINCT

4.5.5 避免在索引列上使用IS NULL和IS NOT NULL

如果索引包含任何可以为空的列，ORACLE将无法使用该索引，举例如下：

低效: (索引失效)

SELECT …

FROM DEPARTMENT

WHERE DEPT\_CODE IS NOT NULL;

高效: (索引有效)

SELECT …

FROM DEPARTMENT

WHERE DEPT\_CODE >=0;

4.5.6 用UNION-ALL 替换UNION

当SQL语句需要UNION两个查询结果集合时,这两个结果集合会以UNION-ALL的方式被合并, 然后在输出最终结果前进行排序.

如果用UNION ALL替代UNION, 这样排序就不是必要了. 效率就会因此得到提高.

4.5.7 用UNION替换OR (适用于索引列)

通常情况下, 用UNION替换WHERE子句中的OR将会起到较好的效果. 对索引列使用OR将造成全表扫描. 注意, 以上规则只针对多个索引列有效. 如果有column没有被索引, 查询效率可能会因为你没有选择OR而降低.

4.5.8 用WHERE替代ORDER BY

ORDER BY 子句只在两种严格的条件下使用索引.

ORDER BY中所有的列必须包含在相同的索引中并保持在索引中的排列顺序.

ORDER BY中所有的列必须定义为非空.

WHERE子句使用的索引和ORDER BY子句中所使用的索引不能并列.

4.5.9 用WHERE子句替换HAVING子句

避免使用HAVING子句, HAVING 只会在检索出所有记录之后才对结果集进行过滤. 这个处理需要排序,总计等操作. 如果能通过WHERE子句限制记录的数目,那就能减少这方面的开销.

例如:

低效:

SELECT REGION，AVG(LOG\_SIZE)

FROM LOCATION

GROUP BY REGION

HAVING REGION REGION != ‘SYDNEY’

AND REGION != ‘PERTH’

高效

SELECT REGION，AVG(LOG\_SIZE)

FROM LOCATION

WHERE REGION REGION != ‘SYDNEY’

AND REGION != ‘PERTH’

GROUP BY REGION

HAVING 中的条件一般用于对一些集合函数的比较,如COUNT() 等等. 除此而外,一般的条件应该写在WHERE子句中.

4.5.10 GROUP BY的使用

提高GROUP BY 语句的效率, 可以通过将不需要的记录在GROUP BY 之前过滤掉.下面两个查询返回相同结果但第二个明显就快了许多.

4.6 全局参数

4.7 其它因素

4.7.1 减少访问数据库的次数

当执行每条SQL语句时, ORACLE在内部执行了许多工作:

? 解析SQL语句

? 估算索引的利用率

? 绑定变量

? 读数据块等等

由此可见, 减少访问数据库的次数 , 就能实际上减少ORACLE的工作量。

4.7.2 常使用COMMIT

只要有可能,在程序中尽量多使用COMMIT, 这样程序的性能得到提高,需求也会因为COMMIT所释放的资源而减少。COMMIT所释放的资源:

ü 回滚段上用于恢复数据的信息.

ü 被程序语句获得的锁

ü redo log buffer 中的空间

ü ORACLE为管理上述3种资源中的内部花费

4.8 常见优化工具介绍

4.8.1 用Explain Plan分析SQL语句

EXPLAIN PLAN 是一个很好的分析SQL语句的工具, 它甚至可以在不执行SQL的情况下分析语句. 通过分析, 我们就可以知道ORACLE是怎么样连接表, 使用什么方式扫描表(索引扫描或全表扫描)以及使用到的索引名称.

4.8.2 SQL PLUS的TRACE

5. 常用术语

5.1 全表扫描

全表扫描就是顺序地访问表中每条记录，ORACLE采用一次读入多个数据块(database block)的方式优化全表扫描

5.2 通过索引访问（ROWID ）

ROWID包含了表中记录的物理位置信息，ORACLE采用索引实现了数据和存放数据的物理位置(ROWID)之间的联系，通常索引提供了快速访问ROWID的方法，因此那些基于索引列的查询就可以得到性能上的提高。

5.3 共享SQL

? Oracle提供对执行过的SQL语句进行高速缓冲的机制。被解析过并且确定了执行路径的SQL语句存放在SGA的共享池中。

? Oracle执行一个SQL语句之前每次先从SGA共享池中查找是否有缓冲的SQL语句，如果有则直接执行该SQL语句。

? 可以通过适当调整SGA共享池大小来达到提高Oracle执行性能的目的。

5.4 ORACLE优化器

? RULE（基于规则）

? COST（基于成本）

? CHOOSE（选择性）

# Session和cookie

cookie用于交互时存放在客户端，即使用你临时文件夹中不存在cookie,但在你的浏览器进程中会临时保存你的cookie!session是交互时存放在服务端，即使用不保存，也在服务进程中。

如果你对网页有交互，服务器如何在众多请求中能识别你那是曾经的哪一个？这依赖于你传递上来的cookie,即合没有任何其他的的交互，在你浏览器进程中也必须保存诸如sessionID之类的cookie!但这个是临时的，只是为了识别你到底是谁而已。知道了你是谁，服务器还需要知道该怎么做，那么在服务器进程中必须存在一个sessionID，这个与你请求的相对应，然后根据这个才知道你是谁，该怎么做。

sessionID是你初次请求时由系统生成，随网页流保存在你的浏览器进程中，以便你在使用postback等回传功能时能识别你！要不然，你回传了，另外一个请求却得到你的回传反应，这有点说不过去吧？

sessionid相当于浏览器与服务进程进行了一个简单的约定，可以理解为初次服务器发给你的一个通行号码，以后你与服务器的任何交互都依赖于这个号码！

而其他的需要长期保存的一些信息也在cookie中，如用户名与密码等等，与这个通信的结果是相同的。也就是说cookie与session同时存在，分别在客户端与服务器！

如果你通过网络嗅探或是其他方式，得到了某一个浏览器正在交互的sessionid以及一些进程中保存的session信息，这个信息在客户端称cookie,这服务器称session。那么你可以利用这些信息进行攻击。

如，在某台电脑中保存有某用户的user与password通行信息时，你可以将自己的cookie违装成目标的cookie，然后可以进行登陆，这种攻击方式叫cookie攻击！

如果他的这些信息保存在了浏览器进程中，也可以伪造，这种其实也是cookie攻击，但由于其不确定性（你还必须拿到sessionid),这种称为session攻击。其实说白了，这种方式在服务器中直接使用的session[“user"]之类的方式取得的，所以伪造时连同sessionid一块伪造，所以才被称为session攻击的。

由于多标签浏览器的存在，还可以进行网页交叉攻击！

# Junit和spring

junit3

就加载junit3对应的jar包就可以

测试类需要继承junit.framework.TestCase

重写setup方法在setup加载需要的配置文件

测试方法需要以test开头testMyCase()

junit4

依赖的jar：org.hamcrest.core.jar junit4.jar

只需要测试方法加org.junit.Test注解

或者按照3的规范继承junit.framework.TestCase并规范命名测试方法即可

和spring的整合

参考：http://blog.csdn.net/mumuzhu2011/article/details/7704120

依赖org.springframework.test.jar包

@org.junit.runner.RunWith(SpringJUnit4ClassRunner.class)

@org.springframework.test.context.ContextConfiguration

(locations = "classpath:applicationContext.xml")

public class ConsumerTest extends org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests

{

@Resource

IConsumerDAO consumerDAOImpl;

//@Test

public void testAbc() {

System.out.println(consumerDAOImpl.getConsumer(1).getUserName());

}

}

手动加载spring配置文件 http://jie66989.iteye.com/blog/1699622

# Jdk新特性

## 1.5

1.自动装箱与拆箱：

2.枚举(常用来设计单例模式)

3.静态导入

4.可变参数

5.内省

## 1.6

1.Web服务元数据

2.脚本语言支持

3.JTable的排序和过滤

4.更简单,更强大的JAX-WS

5.轻量级Http Server

6.嵌入式数据库 Derby

## 1.7

1，switch中可以使用字串了

2.运用List<String> tempList = new ArrayList<>(); 即泛型实例化类型自动推断

3.语法上支持集合，而不一定是数组

4.新增一些取环境信息的工具方法

5.Boolean类型反转，空指针安全,参与位运算

6.两个char间的equals

7.安全的加减乘除

8.map集合支持并发请求，且可以写成 Map map = {name:"xxx",age:18};

## 1.8

1. 允许在接口中有默认方法实现

2. Lambda表达式

3. 函数式接口

4. 方法和构造函数引用

5. Lambda的范围

6. 内置函数式接口

7. Streams

8. Parallel Streams

9. Map

10. 时间日期API

11. Annotations

## 1.9

1. Jigsaw 项目;模块化源码

2. 简化进程API

3. 轻量级 JSON API

4. 钱和货币的API

5. 改善锁争用机制

6. 代码分段缓存

7. 智能Java编译, 第二阶段

8. HTTP 2.0客户端

9. Kulla计划: Java的REPL实现

# Jdk, cglib动态代理

JAVA的动态代理

首先动态代理是设计模式当中代理模式的一种。

首先介绍下代理模式，再介绍Java中的动态代理实践。

1，什么是代理模式？

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。

2，代理模式有什么好处？

在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

3，代理模式一般涉及到的角色有：

抽象角色：声明真实对象和代理对象的共同接口,这样一来在任何可以使用目标对象的地方都可以使用代理对象。

代理角色：代理对象内部含有目标对象的引用，从而可以在任何时候操作目标对象；代理对象提供一个与目标对象相同的接口，以便可以在任何时候替代目标对象。代理对象通常在客户端调用传递给目标对象之前或之后，执行某个操作，而不是单纯地将调用传递给目标对象，同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

真实角色：定义了代理对象所代表的目标对象，代理角色所代表的真实对象，是我们最终要引用的对象,定义了代理对象所代表的目标对象。

代理模式   
代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。   
按照代理的创建时期，代理类可以分为两种。   
静态代理：由程序员创建或特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class文件就已经存在了。   
动态代理：在程序运行时，运用反射机制动态创建而成。

首先看一下静态代理：   
1、Count.java

Java代码

1. package net.battier.dao;
3. /\*\*
4. \* 定义一个账户接口
5. \*
6. \* @author Administrator
7. \*
8. \*/
9. public interface Count {
10. // 查看账户方法
11. public void queryCount();
13. // 修改账户方法
14. public void updateCount();
16. }

2、CountImpl.java

Java代码

1. package net.battier.dao.impl;
3. import net.battier.dao.Count;
5. /\*\*
6. \* 委托类(包含业务逻辑)
7. \*
8. \* @author Administrator
9. \*
10. \*/
11. public class CountImpl implements Count {
13. @Override
14. public void queryCount() {
15. System.out.println("查看账户方法...");
17. }
19. @Override
20. public void updateCount() {
21. System.out.println("修改账户方法...");
23. }
25. }
27. 、CountProxy.java
28. package net.battier.dao.impl;
30. import net.battier.dao.Count;
32. /\*\*
33. \* 这是一个代理类（增强CountImpl实现类）
34. \*
35. \* @author Administrator
36. \*
37. \*/
38. public class CountProxy implements Count {
39. private CountImpl countImpl;
41. /\*\*
42. \* 覆盖默认构造器
43. \*
44. \* @param countImpl
45. \*/
46. public CountProxy(CountImpl countImpl) {
47. this.countImpl = countImpl;
48. }
50. @Override
51. public void queryCount() {
52. System.out.println("事务处理之前");
53. // 调用委托类的方法;
54. countImpl.queryCount();
55. System.out.println("事务处理之后");
56. }
58. @Override
59. public void updateCount() {
60. System.out.println("事务处理之前");
61. // 调用委托类的方法;
62. countImpl.updateCount();
63. System.out.println("事务处理之后");
65. }
67. }

3、TestCount.java

Java代码

1. package net.battier.test;
3. import net.battier.dao.impl.CountImpl;
4. import net.battier.dao.impl.CountProxy;
6. /\*\*
7. \*测试Count类
8. \*
9. \* @author Administrator
10. \*
11. \*/
12. public class TestCount {
13. public static void main(String[] args) {
14. CountImpl countImpl = new CountImpl();
15. CountProxy countProxy = new CountProxy(countImpl);
16. countProxy.updateCount();
17. countProxy.queryCount();
19. }
20. }

观察代码可以发现每一个代理类只能为一个接口服务，这样一来程序开发中必然会产生过多的代理，而且，所有的代理操作除了调用的方法不一样之外，其他的操作都一样，则此时肯定是重复代码。解决这一问题最好的做法是可以通过一个代理类完成全部的代理功能，那么此时就必须使用动态代理完成。   
再来看一下动态代理：   
JDK动态代理中包含一个类和一个接口：   
InvocationHandler接口：   
public interface InvocationHandler {   
public Object invoke(Object proxy,Method method,Object[] args) throws Throwable;   
}   
参数说明：   
Object proxy：指被代理的对象。   
Method method：要调用的方法   
Object[] args：方法调用时所需要的参数

可以将InvocationHandler接口的子类想象成一个代理的最终操作类，替换掉ProxySubject。

Proxy类：   
Proxy类是专门完成代理的操作类，可以通过此类为一个或多个接口动态地生成实现类，此类提供了如下的操作方法：   
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,   
InvocationHandler h)   
                               throws IllegalArgumentException   
参数说明：   
ClassLoader loader：类加载器   
Class<?>[] interfaces：得到全部的接口   
InvocationHandler h：得到InvocationHandler接口的子类实例

Ps:类加载器   
在Proxy类中的newProxyInstance（）方法中需要一个ClassLoader类的实例，ClassLoader实际上对应的是类加载器，在Java中主要有一下三种类加载器;   
Booststrap ClassLoader：此加载器采用C++编写，一般开发中是看不到的；   
Extendsion ClassLoader：用来进行扩展类的加载，一般对应的是jre\lib\ext目录中的类;   
AppClassLoader：(默认)加载classpath指定的类，是最常使用的是一种加载器。

动态代理   
与静态代理类对照的是动态代理类，动态代理类的字节码在程序运行时由Java反射机制动态生成，无需程序员手工编写它的源代码。动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为Java 反射机制可以生成任意类型的动态代理类。java.lang.reflect 包中的Proxy类和InvocationHandler 接口提供了生成动态代理类的能力。

动态代理示例:   
1、BookFacade.java

Java代码

1. package net.battier.dao;
3. public interface BookFacade {
4. public void addBook();
5. }

2、BookFacadeImpl.java

Java代码

1. package net.battier.dao.impl;
3. import net.battier.dao.BookFacade;
5. public class BookFacadeImpl implements BookFacade {
7. @Override
8. public void addBook() {
9. System.out.println("增加图书方法。。。");
10. }
12. }
14. 、BookFacadeProxy.java
16. package net.battier.proxy;
18. import java.lang.reflect.InvocationHandler;
19. import java.lang.reflect.Method;
20. import java.lang.reflect.Proxy;
22. /\*\*
23. \* JDK动态代理代理类
24. \*
25. \* @author student
26. \*
27. \*/
28. public class BookFacadeProxy implements InvocationHandler {
29. private Object target;
30. /\*\*
31. \* 绑定委托对象并返回一个代理类
32. \* @param target
33. \* @return
34. \*/
35. public Object bind(Object target) {
36. this.target = target;
37. //取得代理对象
38. return Proxy.newProxyInstance(target.getClass().getClassLoader(),
39. target.getClass().getInterfaces(), this);   //要绑定接口(这是一个缺陷，cglib弥补了这一缺陷)
40. }
42. @Override
43. /\*\*
44. \* 调用方法
45. \*/
46. public Object invoke(Object proxy, Method method, Object[] args)
47. throws Throwable {
48. Object result=null;
49. System.out.println("事物开始");
50. //执行方法
51. result=method.invoke(target, args);
52. System.out.println("事物结束");
53. return result;
54. }
56. }

3、TestProxy.java

Java代码

1. package net.battier.test;
3. import net.battier.dao.BookFacade;
4. import net.battier.dao.impl.BookFacadeImpl;
5. import net.battier.proxy.BookFacadeProxy;
7. public class TestProxy {
9. public static void main(String[] args) {
10. BookFacadeProxy proxy = new BookFacadeProxy();
11. BookFacade bookProxy = (BookFacade) proxy.bind(new BookFacadeImpl());
12. bookProxy.addBook();
13. }
15. }

但是，JDK的动态代理依靠接口实现，如果有些类并没有实现接口，则不能使用JDK代理，这就要使用cglib动态代理了。

Cglib动态代理   
JDK的动态代理机制只能代理实现了接口的类，而不能实现接口的类就不能实现JDK的动态代理，cglib是针对类来实现代理的，他的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类进行代理。   
示例   
1、BookFacadeCglib.java

Java代码

1. package net.battier.dao;
3. public interface BookFacade {
4. public void addBook();
5. }

2、BookCadeImpl1.java

Java代码

1. package net.battier.dao.impl;
3. /\*\*
4. \* 这个是没有实现接口的实现类
5. \*
6. \* @author student
7. \*
8. \*/
9. public class BookFacadeImpl1 {
10. public void addBook() {
11. System.out.println("增加图书的普通方法...");
12. }
13. }

3、BookFacadeProxy.java

Java代码

1. package net.battier.proxy;
3. import java.lang.reflect.Method;
5. import net.sf.cglib.proxy.Enhancer;
6. import net.sf.cglib.proxy.MethodInterceptor;
7. import net.sf.cglib.proxy.MethodProxy;
9. /\*\*
10. \* 使用cglib动态代理
11. \*
12. \* @author student
13. \*
14. \*/
15. public class BookFacadeCglib implements MethodInterceptor {
16. private Object target;
18. /\*\*
19. \* 创建代理对象
20. \*
21. \* @param target
22. \* @return
23. \*/
24. public Object getInstance(Object target) {
25. this.target = target;
26. Enhancer enhancer = new Enhancer();
27. enhancer.setSuperclass(this.target.getClass());
28. // 回调方法
29. enhancer.setCallback(this);
30. // 创建代理对象
31. return enhancer.create();
32. }
34. @Override
35. // 回调方法
36. public Object intercept(Object obj, Method method, Object[] args,
37. MethodProxy proxy) throws Throwable {
38. System.out.println("事物开始");
39. proxy.invokeSuper(obj, args);
40. System.out.println("事物结束");
41. return null;

44. }
46. }

4、TestCglib.java

Java代码

1. package net.battier.test;
3. import net.battier.dao.impl.BookFacadeImpl1;
4. import net.battier.proxy.BookFacadeCglib;
6. public class TestCglib {
8. public static void main(String[] args) {
9. BookFacadeCglib cglib=new BookFacadeCglib();
10. BookFacadeImpl1 bookCglib=(BookFacadeImpl1)cglib.getInstance(new BookFacadeImpl1());
11. bookCglib.addBook();
12. }
13. }

# Maven和ant

Ant是软件构建工具，Maven的定位是软件项目管理和理解工具。Maven除了具备Ant的功能外，还增加了以下主要的功能：

1）使用Project Object Model来对软件项目管理；

2）内置了更多的隐式规则，使得构建文件更加简单；

3）内置依赖管理和Repository来实现依赖的管理和统一存储；

4）内置了软件构建的生命周期；

1. 创建Maven的普通java项目：   
   mvn archetype:create   
   -DgroupId=packageName   
   -DartifactId=projectName    
2. 创建Maven的Web项目：     
    mvn archetype:create   
    -DgroupId=packageName      
    -DartifactId=webappName   
    -DarchetypeArtifactId=maven-archetype-webapp      
3. 编译源代码： mvn compile   
4. 编译测试代码：mvn test-compile      
5. 运行测试：mvn test     
6. 产生site：mvn site     
7. 打包：mvn package     
8. 在本地Repository中安装jar：mvn install   
9. 清除产生的项目：mvn clean     
10. 生成eclipse项目：mvn eclipse:eclipse    
11. 生成idea项目：mvn idea:idea    
12. 组合使用goal命令，如只打包不测试：mvn -Dtest package     
13. 编译测试的内容：mvn test-compile    
14. 只打jar包: mvn jar:jar    
15. 只测试而不编译，也不测试编译：mvn test -skipping compile -skipping test-compile   
      ( -skipping 的灵活运用，当然也可以用于其他组合命令)    
16. 清除eclipse的一些系统设置:mvn eclipse:clean

首先，Nexus是一种远程仓库，根据上段的介绍，我们已经知道远程仓库的作用。在远程仓库中，默认的是中央仓库，中央仓库是Maven核心自带的远程仓库。那就使用中央仓库不就得了吗，为什么我们要安装Nexus呢？

我们从项目实际开发来看：

1.一些无法从外部仓库下载的构件，例如内部的项目还能部署到私服上，以便供其他依赖项目使用。

2. 为了节省带宽和时间，在局域网内架设一个私有的仓库服务器，用其代理所有外部的远程仓库。当本地Maven项目需要下载构件时，先去私服请求，如果私服没有，则再去远程仓库请求，从远程仓库下载构件后，把构件缓存在私服上。这样，及时暂时没有Internet链接，由于私服已经缓存了大量构件，整个项目还是可以正常使用的。同时，也降低了中央仓库的负荷。

# Mybatis

通过上篇介绍mybatis与hibernate区别，我们已经能得出一些mybatis的优缺点，但那只是相对于hibernate的，并不全面，我来继续总结mybatis的优缺点，以便大家对于mybatis的了解能更全面些。但我所说的优缺点，仅是我个人总结并结合使用体验后得出的结果，并不能代表大众想法，因此才以“浅谈”作为文章标题。如果大家的见解与我不同，欢迎积极提出来一块讨论，我也借以弥补自己认识的不足和短见。

优点：

1. 易于上手和掌握。

2. sql写在xml里，便于统一管理和优化。

3. 解除sql与程序代码的耦合。

4. 提供映射标签，支持对象与数据库的orm字段关系映射

5. 提供对象关系映射标签，支持对象关系组建维护

6. 提供xml标签，支持编写动态sql。

缺点：

1. sql工作量很大，尤其是字段多、关联表多时，更是如此。

2. sql依赖于数据库，导致数据库移植性差。

3. 由于xml里标签id必须唯一，导致DAO中方法不支持方法重载。

4. 字段映射标签和对象关系映射标签仅仅是对映射关系的描述，具体实现仍然依赖于sql。（比如配置了一对多Collection标签，如果sql里没有join子表或查询子表的话，查询后返回的对象是不具备对象关系的，即Collection的对象为null）

5. DAO层过于简单，对象组装的工作量较大。

6. 不支持级联更新、级联删除。

7. 编写动态sql时,不方便调试，尤其逻辑复杂时。

8 提供的写动态sql的xml标签功能简单（连struts都比不上），编写动态sql仍然受限，且可读性低。

9. 不查询主键字段，容易造成查询出的对象有“覆盖”现象。

10. 参数的数据类型支持不完善。（如参数为Date类型时，容易报没有get、set方法，需在参数上加@param）

11. 多参数时，使用不方便，功能不够强大。（目前支持的方法有map、对象、注解@param以及默认采用012索引位的方式）

12. 缓存使用不当，容易产生脏数据。

总结：

mybatis的优点其实也是mybatis的缺点，正因为mybatis使用简单，数据的可靠性、完整性的瓶颈便更多依赖于程序员对sql的使用水平上了。sql写在xml里，虽然方便了修改、优化和统一浏览，但可读性很低，调试也非常困难，也非常受限，无法像jdbc那样在代码里根据逻辑实现复杂动态sql拼接。mybatis简单看就是提供了字段映射和对象关系映射的jdbc，省去了数据赋值到对象的步骤而已，除此以外并无太多作为，不要把它想象成hibernate那样强大，简单小巧易用上手，方便浏览修改sql就是它最大的优点了。

mybatis适用于小型且程序员能力较低的项目和人群使用，对于中大型项目来说我并不推荐使用，如果觉得hibernate效率低的话（实际上也是使用不当所致，hibernate是实际上是不适用于拥有高负载的工程项目），还不如直接用spring提供的jdbc简单框架（Template），同样支持对象映射。

# Spring

大部分项目都少不了Spring的身影，为什么大家对他如此青睐，而且对他的追捧丝毫没有减退之势呢

Spring是什么：

Spring是一个轻量级的DI和AOP容器框架。

说它轻量级有一大部分原因是相对与EJB的（虽然本人从没有接触过EJB的应用），重要的是，Spring是非侵入式的，基于spring开发的应用一般不依赖于spring的类。

DI:称作依赖注入(Dependency Injection),和控制反转一个概念，具体的讲，当一个角色需要另外一个角色协助的时候，在传统的程序设计中，通常有调用者来创建被调用者的实例。但是在spring中创建被调用者将不再有调用者完成，因此叫控制反转。创建被调用对象有Spring来完成，在容器实例化对象的时候主动的将被调用者（或者说它的依赖对象）注入给调用对象，因此又叫依赖注入。

AOP：Spring对面向切面编程提供了强有力的支持，通过它让我们将业务逻辑从应用服务（如事务管理）中分离出来，实现了高内聚开发，应用对象只关注业务逻辑，不再负责其它系统问题（如日志、事务等）。Spring支持用户自定义切面。

面向切面编程是面向对象编程的有力补充。面向对象编程将程序分成各个层次的对象，面向切面的程序将运行过程分解成各个切面。AOP是从运行程序的角度去考虑程序的结构，提取业务处理过程的切面，OOP是静态的抽象，AOP是动态的抽象，是对应用执行过程的步骤进行抽象，从而获得步骤之间的逻辑划分。

容器：Spring是个容器，因为它包含并且管理应用对象的生命周期和配置。如对象的创建、销毁、回调等。

框架：Spring作为一个框架，提供了一些基础功能，（如事务管理，持久层集成等），使开发人员更专注于开发应用逻辑。

看完了Spring是什么，再来看看Spring有哪些优点

1.使用Spring的IOC容器，将对象之间的依赖关系交给Spring，降低组件之间的耦合性，让我们更专注于应用逻辑

2.可以提供众多服务，事务管理，WS等。

3.AOP的很好支持，方便面向切面编程。

4.对主流的框架提供了很好的集成支持，如Hibernate,Struts2,JPA等

5.Spring DI机制降低了业务对象替换的复杂性。

6.Spring属于低侵入，代码污染极低。

7.Spring的高度可开放性，并不强制依赖于Spring，开发者可以自由选择Spring部分或全部

# 序列化

Java中，一切都是对象，在分布式环境中经常需要将Object从这一端网络或设备传递到另一端。这就需要有一种可以在两端传输数据的协议。Java序列化机制就是为了解决这个问题而产生。如果想要进行网络传输数据或者将数据保存到本个地文件里就必须将它序列化

# 反射机制

java的反射机制就是增加程序的灵活性，避免将程序写死到代码里，

例如： 实例化一个 person()对象， 不使用反射， new person(); 如果想变成 实例化 其他类， 那么必须修改源代码，并重新编译。

使用反射： class.forName("person").newInstance()； 而且这个类描述可以写到配置文件中，如 \*\*.xml, 这样如果想实例化其他类，只要修改配置文件的"类描述"就可以了，不需要重新修改代码并编译。

复制过来的：http://zhidao.baidu.com/question/478254128.html

增加程序的灵活性。

如struts中。请求的派发控制。

当请求来到时。struts通过查询配置文件。找到该请求对应的action。已经方法。

然后通过反射实例化action。并调用响应method。

如果不适用反射，那么你就只能写死到代码里了。

所以说，一个灵活，一个不灵活。

很少情况下是非用反射不可的。大多数情况下反射是为了提高程序的灵活性。

因此一般框架中使用较多。因为框架要适用更多的情况。对灵活性要求较高。

# 泛型

1，类型安全。 泛型的主要目标是提高 Java 程序的类型安全。通过知道使用泛型定义的变量的类型限制，编译器可以在一个高得多的程度上验证类型假设。没有泛型，这些假设就只存在于程序员的头脑中（或者如果幸运的话，还存在于代码注释中）。

2，消除强制类型转换。 泛型的一个附带好处是，消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。

3，潜在的性能收益。 泛型为较大的优化带来可能。在泛型的初始实现中，编译器将强制类型转换（没有泛型的话，程序员会指定这些强制类型转换）插入生成的字节码中。但是更多类型信息可用于编译器这一事实，为未来版本的 JVM 的优化带来可能。由于泛型的实现方式，支持泛型（几乎）不需要 JVM 或类文件更改。所有工作都在编译器中完成，编译器生成类似于没有泛型（和强制类型转换）时所写的代码，只是更能确保类型安全而已。

# 注解

Java 注解，从名字上看是注释，解释。但功能却不仅仅是注释那么简单。注解（Annotation） 为我们在代码中添加信息提供了一种形式化的方法，是我们可以在稍后某个时刻方便地使用这些数据（通过 解析注解 来使用这些数据），常见的作用有以下几种：

•生成文档。这是最常见的，也是java 最早提供的注解。常用的有@see @param @return 等

•跟踪代码依赖性，实现替代配置文件功能。比较常见的是spring 2.5 开始的基于注解配置。作用就是减少配置。现在的框架基本都使用了这种配置来减少配置文件的数量。也是

•在编译时进行格式检查。如@Override 放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出。

# Socket

对于Java Socket编程而言，有两个概念，一个是ServerSocket，一个是Socket。服务端和客户端之间通过Socket建立连接，之后它们就可以进行通信了。首先ServerSocket将在服务端监听某个端口，当发现客户端有Socket来试图连接它时，它会accept该Socket的连接请求，同时在服务端建立一个对应的Socket与之进行通信。这样就有两个Socket了，客户端和服务端各一个。

       对于Socket之间的通信其实很简单，服务端往Socket的输出流里面写东西，客户端就可以通过Socket的输入流读取对应的内容。Socket与Socket之间是双向连通的，所以客户端也可以往对应的Socket输出流里面写东西，然后服务端对应的Socket的输入流就可以读出对应的内容。下面来看一些服务端与客户端通信的例子：

# 大数据和Hadoop专题

大数据目前分为四大块：大数据技术、大数据工程、大数据科学和大数据应用。其中云计算是属于大数据技术的范畴，是一种通过Internet以服务 的方式提供动态可伸缩的虚拟化的资源的计算模式。那么这种计算模式如何实现呢，Hadoop的来临解决了这个问题，Hadoop是Apache（阿帕切） 的一个开源项目，它是一个对大量数据进行分布式处理的软件架构，在这个架构下组织的成员HDFS（Hadoop分布式文件系统），MapReduce、 Hbase 、Zookeeper（一个针对大型分布式系统的可靠协调系统），hive（基于Hadoop的一个数据仓库工具）等。

1.云计算属于大数据中的大数据技术范畴。

2.云计算包含大数据。

3.云和大数据是两个领域。

# JSON和XML比较

1.在可读性方面，JSON和XML的数据可读性基本相同。JSON和XML的可读性可谓不相上下，一边是建议的语法，一边是规范的标签形式，很难分出胜负。

2.在可扩展性方面，XML天生有很好的扩展性，JSON当然也有，没有什么是XML能扩展，JSON不能的。

3.在编码难度方面，XML有丰富的编码工具，比如Dom4j、JDom等，JSON也有json.org提供的工具，但是JSON的编码明显比XML容易许多，即使不借助工具也能写出JSON的代码，可是要写好XML就不太容易了。

4.在解码难度方面，XML的解析得考虑子节点父节点，让人头昏眼花，而JSON的解析难度几乎为0。这一点XML输的真是没话说。

5.在流行度方面，XML已经被业界广泛的使用，而JSON才刚刚开始，但是在Ajax这个特定的领域，未来的发展一定是XML让位于JSON。到时Ajax应该变成Ajaj(AsynchronousJavascript and JSON)了。

6.JSON和XML同样拥有丰富的解析手段。

7.JSON相对于XML来讲，数据的体积小。

8.JSON与JavaScript的交互更加方便。

9.JSON对数据的描述性比XML较差。

10.XML安全性好，JSON扩展脚本攻击。

# 线程安全

线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。 线程不安全就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据

比如一个 ArrayList 类，在添加一个元素的时候，它可能会有两步来完成：1. 在 Items[Size] 的位置存放此元素；2. 增大 Size 的值。

在单线程运行的情况下，如果 Size = 0，添加一个元素后，此元素在位置 0，而且 Size=1；

而如果是在多线程情况下，比如有两个线程，线程 A 先将元素1存放在位置 0。但是此时 CPU 调度线程A暂停，线程 B 得到运行的机会。线程B向此 ArrayList 添加元素2，因为此时 Size 仍然等于 0 （注意，我们假设的是添加一个元素是要两个步骤，而线程A仅仅完成了步骤1），所以线程B也将元素存放在位置0。然后线程A和线程B都继续运行，都增加 Size 的值，结果Size等于2。

那好，我们来看看 ArrayList 的情况，期望的元素应该有2个，而实际只有一个元素，造成丢失元素，而且Size 等于 2。这就是“线程不安全”了。

# Spring Bean

spring单例通过设置属性scope="Singleton"完成（默认），其概念和java中的单例概念一致，如在web应用中，每次请求使用

的是同一个实例对象。但spring的单例仅限制在其上下文环境中，并没有限制你通过其他方式去创建此对象的其他实例。说白

了是spring帮你完成模拟的单例创建，使用。

scope="prototype" 多例，

默认情况为单例方式：scope=”singleton”

singleton

单实例作用域，这是Spring容器默认的作用域，使用singleton作用域生成的是单实例，在整个Bean容器中仅保留一个实例对象供所有调用者共享引用。

单例模式对于那些无会话状态的Bean（如辅助工具类、DAO组件、业务逻辑组件等）是最理想的选择。

prototype

原型模式，这是多实例作用域，针对每次不同的请求，Bean容器均会生成一个全新的Bean实例以供调用者使用。prototype作用域非常适用于那些需要保

持会话状态的Bean实例，有一点值得注意的就是，Spring不能对一个prototype Bean的整个生命周期负责，容器在初始化、装配好一个prototype实例后

，将它交给客户端，随后就对该prototype实例不闻不问了。因此，客户端要负责prototype实例的生命周期管理。

request

针对每次HTTP请求，Spring容器会根据Bean的定义创建一个全新的Bean实例， 且该Bean实例仅在当前HTTP request内有效，因此可以根据需要放心地更

改所建实例的内部状态， 而其他请求中根据Bean定义创建的实例，将不会看到这些特定于某个请求的状态变化。 当处理请求结束，request作用域的Be

an实例将被销毁。该作用域仅在基于web的Spring ApplicationContext情形下有效。

session

针对某个HTTP Session，Spring容器会根据Bean定义创建一个全新的Bean实例，且该Bean实例仅在当前HTTP Session内有效。 与request作用域一样，我

们可以根据需要放心地更改所创建实例的内部状态，而别的HTTP Session中根据Bean定义创建的实例， 将不会看到这些特定于某个HTTP Session的状态

变化。 当HTTP Session最终被废弃的时候，在该HTTP Session作用域内的Bean实例也会被废弃掉。该作用域仅在基于Web的Spring ApplicationContext

情形下有效。

global session

global session作用域类似于标准的HTTP Session作用域，不过它仅仅在基于portlet的Web应用中才有意义。portlet规范定义了全局Session的概念，它

被所有构成某个portlet Web应用的各种不同的portlet所共享。在global session作用域中定义的Bean被限定于全局portlet Session的生命周期范围内

。如果我们是在编写一个标准的基于Servlet的Web应用，并且定义了一个或多个具有global session作用域的Bean，系统会使用标准的HTTP Session作用

域，并且不会引起任何错误。该作用域仅在基于Web的Spring ApplicationContext情形下有效。

# Spring事物配置

配置事务管理器 transactionManager

配置事务传播特性 advice

配置参与事务的类 execution(\* com.test.testAda.test.model.service.\*.\*(..))"

# Spring IOC与New比较

不用手动new一个对象。就是由spring来负责控制对象的生命周期和对象间的关系。

A中需要一个Connection，至于这个Connection怎么构造，何时构造，A不需要知道。在系统运行时，spring会在适当的时

候制造一个Connection，然后像打针一样，注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connec

tion才能正常运行，而这个Connection是由spring注入到A中的，依赖注入的名字就这么来的。

目的降低对象和对象之间的耦合性。

# Spring AOP原理

AOP即Aspect-Oriented Programming的缩写，中文意思是面向切面（或方面）编程。AOP实际上是一种编程思想，可以通过预编译方式和运行期动态

代理实现在不修改源代码的情况下给程序动态统一添加功能的一种思想。

在传统的面向对象（Object-Oriented Progr amming，OOP）编程中，我们总是按照某种特定的执行顺序来实现业务流程，各个执行步骤之间是相互

衔接、相互耦合的，对垂直切面关注度很高，横切面关注却很少，也很难关注。那么怎样可以解决这个问题呢？我们需要AOP，关注系统的“切面”，在

适当的时候“拦截”程序的执行流程，把程序的预处理和后处理交给某个拦截器来完成。这样，业务流程就完全的从其它无关的代码中解放出来，各模

块之间的分工更加明确，程序维护也变得容易多了。

正如上所说，AOP不是一种技术，而是编程思想。凡是符合AOP思想的技术，都可以看成是AOP的实现。目前的AOP实现有AspectJ、JBoss4.0、nanning

、spring等项目。其中Spring对AOP进行了很好的实现，同时Spring AOP也是Spring的两大核心一。

AOP的发展目前已经历了两个阶段：第一阶--静态AOP和第二阶段—动态AOP。静态AOP阶段，相应的横切关注点以Aspect形式实现之后，会通过特定的编

译器，将实现后的Aspect编译并织入到系统的静态类中。比如AspectJ会使用ajc编译器将各个Aspect以Java字节码的形式编译到系统的各个功能模块

中，已达到融合Aspect和Class的目的。动态AOP阶段，AOP的织入过程在系统运行开始之后进行，而不是预先编译到系统中，可以在调整织入点以及织

入逻辑单元的同时，不必变更系统其他的模块，甚至在系统运行的时候，也可以动态更改织入逻辑。这两个阶段也为我们提供了通过预编译方式和运

行期动态代理方式，实现了在不修改源代码的情况下给程序统一添加功能的目的。

AOP主要的意图是：允许通过分离应用的业务逻辑与系统级服务进行内聚性的开发。应用对象只实现业务逻辑即可，并不负责其它的系统级关注点。

AOP主要应用场景是：日志记录、跟踪、监控和优化，性能统计、优化，安全、权限控制，应用系统的异常捕捉及处理，事务处理，缓存，持久化，懒

加载（Lazy loading），内容传递，调试，资源池，同步等等。

AOP的相关概念：

Joinpoint：在系统运行之前，AOP的功能模块都需要织入到OOP的功能模块中。所以，要进行这种织入过程，我们需要知道在系统的那些执行点上进行

织入操作。这些将要在其上进行织入操作的系统执行点就称之为Joinpoint。（在Spring AOP 中仅支持方法级别的Joinpoint）

Pointcut：代表的是Joinpoint的表达方式。将横切逻辑织入当前系统的过程中，需要参照Pointcut规定的Joinpoint信息，才可以知道应该往系统的

哪些Joinpoint上织入横切逻辑。

Pointcut指定系统中符合条件的一组Joinpoint。

Advice：是单一切关注点逻辑的载体，代表将会织入到Joinpoint处的横切逻辑。如果将Aspect比作OOP中的Class，那么Advice就相当于Class中的

Method。按照Advice在Joinpoint处执行时机的差异或者完成功能的不同，Advice可分成以下具体形式：Before Advice,After Advice(Afterreturning

Advice、AfterThrowing Advice、AfterFinally Advice),Around Advice,Introduction。

Aspect：是对体统中的横切关注点逻辑进行模块化封装的AOP概念实体。Aspect可以包含多个Pointcut以及相应的Advice定义。

织入：完成横切关注点逻辑（以Aspect模块化的横切关注点）到OOP系统的过程。

SpringAOP的实现机制：

Spring AOP**采用动态代理机制和字节码生成技术实现**。与第一代AspectJ采用编译器将横切逻辑织入目标对象不同，动态代理机制和字节码生成都是在

运行期间为目标对象生成一个代理对象，而将横切逻辑织入到这个代理对象中，系统最终使用的是织入了横切逻辑的代理对象，而不是真正的目标对象。

默认情况下，如果Spring AOP发现目标对象实现了相应Interface，则采用动态代理机制为其生成代理对象实例。而如果目标对象没有实现任何Interface

，Spring AOP会尝试使用CGLIB(Code Generation Library)这种动态字节码生成类库，为目标对象生成动态的代理对象实例。

SpringAOP中需注意的相应概念实体：

<!--[if !supportLists]--> 1.Around Advice

Spring中没有直接定义对应Around Advice的实现接口，而是直接采用AOP Alliance

的标准接口，即org.aopalliance.intercept.MethodInterceptor,该接口定义如下：

Public interfaceMethodInterceptor extends Interceptor{

Object invoke(MethodInvocation invocation)throws Throwable;

}

通过MethodInterceptor的invoke方法的MethodInvocation参数，我们可以控制对相应的Joinpoint的拦截行为。通过调用MethodInvocation的proceed（）

方法，可以让程序执行继续沿着调用链传播。如果在哪一个MethodInterceptor中没有调用proceed（），那么程序的执行将会在当前MethodInterceptor

处“短路”，Joinpoint上的调用链将被中断，同一Joinpoint上的其他MethodInterceptor的逻辑以及Joinpoint出的方法逻辑将不会被执行。

<!--[if !supportLists]--> 2.Spring AOP中的Aspect

Advisor是Spring中用的Aspect。但是，与正常的Aspect不同，Advisor通常只持有一个Pointcut和一个advice。而理论上，Aspect定义中可以有多个

Pointcut和多个Advice。所以，我们认为advisor是一种特殊的Aspect。

<!--[if !supportLists]--> 3.Spring AOP的织

在Spring AOP中，使用类org.springframework.aop.framework.ProxyFactory作为织入器。在IoC容器中，使用org.springframework.aop.framework.

ProxyFactoryBean作为织入器。

（AspectJ采用ajc编译器作为织入器。JBOSS AOP使用自定义的ClassLoader作为织入器）

AOP（Aspect-OrientedProgramming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。

OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为

的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代

码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明

的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不

利于各个模块的重用。

uthentication 权限

Caching 缓存

Context passing 内容传递

Error handling 错误处理

Lazy loading　懒加载

Debugging　　调试

logging, tracing, profiling and monitoring　记录跟踪　优化　校准

Performance optimization　性能优化

Persistence　　持久化

Resource pooling　资源池

Synchronization　同步

Transactions 事务

**AOP核心概念**

1、横切关注点

对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

2、切面（aspect）

类是对物体特征的抽象，切面就是对横切关注点的抽象

3、连接点（joinpoint）

被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器

4、切入点（pointcut）

对连接点进行拦截的定义

5、通知（advice）

所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知五类

6、目标对象

代理的目标对象

7、织入（weave）

将切面应用到目标对象并导致代理对象创建的过程

8、引入（introduction）

在不修改代码的前提下，引入可以在**运行期**为类动态地添加一些方法或字段

**Spring对AOP的支持**

**Spring中AOP代理由Spring的IOC容器负责生成、管理，其依赖关系也由IOC容器负责管理**。因此，AOP代理可以直接使用容器中的其它bean实例作为目标，这种关系可由IOC容器的依赖注入提供。Spring创建代理的规则为：

1、**默认使用Java动态代理来创建AOP代理**，这样就可以为任何接口实例创建代理了

2、**当需要代理的类不是代理接口的时候，Spring会切换为使用CGLIB代理**，也可强制使用CGLIB

AOP编程其实是很简单的事情，纵观AOP编程，程序员只需要参与三个部分：

1、定义普通业务组件

2、定义切入点，一个切入点可能横切多个业务组件

3、定义增强处理，增强处理就是在AOP框架为普通业务组件织入的处理动作

所以进行AOP编程的关键就是定义切入点和定义增强处理，一旦定义了合适的切入点和增强处理，AOP框架将自动生成AOP代理，即：**代理对象的方法=增强处理+被代理对象**的方法。

强制使用CGLIB生成代理

前面说过Spring使用动态代理或是CGLIB生成代理是有规则的，高版本的Spring会自动选择是使用动态代理还是CGLIB生成代理内容，当然我们也可以强制使用CGLIB生成代理，那就是<aop:config>里面有一个"proxy-target-class"属性，这个属性值如果被设置为true，那么基于类的代理将起作用，如果proxy-target-class被设置为false或者这个属性被省略，那么基于接口的代理将起作用

**通知（Advice）类型**  
前置通知（Before advice） ：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext中在<aop:aspect>里面使用<aop:before>元素进行声明。  
后通知（After advice） ：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext中在<aop:aspect>里面使用<aop:after>元素进行声明。  
返回后通知（After return advice） ：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext中在<aop:aspect>里面使用<after-returning>元素进行声明。  
环绕通知（Around advice） ：包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext中在<aop:aspect>里面使用<aop:around>元素进行声明。  
抛出异常后通知（After throwing advice） ： 在方法抛出异常退出时执行的通知。 ApplicationContext中在<aop:aspect>里面使用<aop:after-throwing>元素进行声明。

切入点表达式 ：如execution(\* com.spring.service.\*.\*(..))

### 特点

1、降低模块之间的耦合度

2、使系统容易扩展

3、更好的代码复用。

# Spring MVC

遵循MVC模式；和Spring很好整合，采用注解机制开发简便，

Spring MVC属于SpringFrameWork的后续产品，已经融合在Spring Web Flow里面。Spring 框架提供了构建 Web 应用程序的全功能

它是一个典型的教科书式的mvc构架，而不像struts等都是变种或者不是完全基于mvc系统的框架，对于初学者或

者想了解mvc的人来说我觉得 spring是最好的，它的实现就是教科书！第二它和tapestry一样是一个纯正的servlet系统，

这也是它和tapestry相比 struts所没有的优势。而且框架本身有代码，而且看起来容易理解。

1:spring3开发效率高于struts

2:spring3 mvc可以认为已经100%零配置

3:struts2是类级别的拦截， 一个类对应一个request上下文，springmvc是方法级别的拦截，

一个方法对应一个request上下文，而方法同时又跟一个url对应所以说从架构本身上 spring3

mvc就容易实现restful url 而struts2的架构实现起来要费劲因为struts2 action的一个方法可以对应

一个url而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了

4:spring3mvc的方法之间基本上独立的，独享request response数据请求数据通过参数获取，处理结果

通过ModelMap交回给框架方法之间不共享变量而struts2搞的就比较乱，虽然方法之间也是独立的，但其

所有Action变量是共享的这不会影响程序运行，却给我们编码 读程序时带来麻烦

5:由于Struts2需要针 对每个Request进行封装，把Request，Session等Servlet生命周期的变量封

装成一个一个Map，供给每个Action使用，并保证线程安全。所以在原则上，是比较耗费内存的

# 分布式架构/分布式事务/分布式数据库/分布式缓存

分布式服务化治理框架（微服务框架），久经阿里电商平台的大规模复杂业务的高并发考验，到目前为止Dubbo仍然是开源界中体系相当完善的服务化治理框架。

《基于Dubbo的分布式系统架构实战》以Dubbo为核心框架，基于真实的第三方支付项目系统架构实战经验而形成的一整套分布式服务化系统架构技术解决方案。

分布式系统架构简单的说是运行在多个处理器上的软件构架设计。

1.分布式系统是建立在网络之上的软件系统。正是因为软件的特性，所以分布式系统具有高度的内聚性和透明性。

2.网络和分布式系统之间的区别更多的在于高层软件（特别是操作系统），而不是硬件。内聚性是指每一个数据库分布节点高度自治，有本地的数据库管理系统。

3.架构，又名软件架构，是有关软件整体结构与组件的抽象描述，用于指导大型软件系统各个方面的设计。架构描述语言（ADL）用于描述软件的体系架构。

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。以上是百度百科的解释，简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

分布式事务怎么处理：

"目前比较多的解决方案有几个：

一、结合MQ消息中间件实现的可靠消息最终一致性

二、TCC补偿性事务解决方案

三、最大努力通知型方案

第一种方案：可靠消息最终一致性，需要业务系统结合MQ消息中间件实现，在实现过程中需要保证消息的成功发送及成功消费。即需要通过业务系统控制MQ的消息状态

第二种方案：TCC补偿性，分为三个阶段TRYING-CONFIRMING-CANCELING。每个阶段做不同的处理。

TRYING阶段主要是对业务系统进行检测及资源预留

CONFIRMING阶段是做业务提交，通过TRYING阶段执行成功后，再执行该阶段。默认如果TRYING阶段执行成功，CONFIRMING就一定能成功。

CANCELING阶段是回对业务做回滚，在TRYING阶段中，如果存在分支事务TRYING失败，则需要调用CANCELING将已预留的资源进行释放。

第三种方案：最大努力通知xing型，这种方案主要用在与第三方系统通讯时，比如：调用微信或支付宝支付后的支付结果通知。这种方案也是结合MQ进行实现，例如：通过MQ发送http请求，设置最大通知次数。达到通知次数后即不再通知。

具体的案例你也可以参考下这篇博客，它上面有完整的电商系统分布式事务实现案例：

目前，服务器的磁盘和内存，cpu都相对较好，一台数据库服务器可以存储好几亿条的数据，在一个什么样的情况下，应该考虑分布式数据库的，百亿？千亿？

考虑用分布式数据库，肯定是容量或者性能方面，现有的单机数据库满足不了业务的需求。当然，遇到了容量或者性能的问题，也不一定要用分布式数据库，可以通过scale-up的方式，即升级数据库服务器的CPU、内存、磁盘，将SATA/SAS盘换SSD盘等方式解决。不过相对scale-up来说， 分布式数据库这种scale-out的方式，扩展性会更强一些，一般来说也更具性价比。

随着传统的数据库技术日趋成熟、计算机网络技术的飞速发展和应用范围的扩大，以分布式为主要特征的数据库系统的研究与开发受到人们的注意。分布式数据库是数据库技术与网络技术相结合的产物，在数据库领域已形成一个分支。分布式数据库的研究始于20世纪70年代中期。世界上第一个分布式数据库系统SDD-1是由美国计算机公司（CCA）于1979年在DEC计算机上实现。20世纪90年代以来，分布式数据库系统进入商品化应用阶段，传统的关系数据库产品均发展成以计算机网络及多任务操作系统为核心的分布式数据库产品，同时分布式数据库逐步向客户机/服务器模式发展

分布式缓存系统是为了解决数据库服务器和web服务器之间的瓶颈。

如果一个网站的流量很大，这个瓶颈将会非常明显，每次数据库查询耗费的时间将会非常可观。

对于更新速度不是很快的网站，我们可以用静态化来避免过多的数据库查询。

对于更新速度以秒计的网站，静态化也不会太理想，可以用缓存系统来构建。

如果只是单台服务器用作缓存，问题不会太复杂，如果有多台服务器用作缓存，就要考虑缓存服务器的负载均衡。

分布式缓存具有如下特性:

1) 高性能:当传统数据库面临大规模数据访问时,磁盘I/O 往往成为性能瓶颈,从而导致过高的响应延迟.分布式缓存将高速内存作为数据对象的存储介质,数据以key/value 形式存储,理想情况下可以获得DRAM 级的读写性能;

2) 动态扩展性:支持弹性扩展,通过动态增加或减少节点应对变化的数据访问负载,提供可预测的性能与扩展性;同时,最大限度地提高资源利用率;

3) 高可用性:可用性包含数据可用性与服务可用性两方面.基于冗余机制实现高可用性,无单点失效(single point of failure),支持故障的自动发现,透明地实施故障切换,不会因服务器故障而导致缓存服务中断或数据丢失.动态扩展时自动均衡数据分区,同时保障缓存服务持续可用;

4) 易用性:提供单一的数据与管理视图;API 接口简单,且与拓扑结构无关;动态扩展或失效恢复时无需人工配置;自动选取备份节点;多数缓存系统提供了图形化的管理控制台,便于统一维护;

5) 分布式代码执行(distributed code execution):将任务代码转移到各数据节点并行执行,客户端聚合返回结果,从而有效避免了缓存数据的移动与传输.最新的Java 数据网格规范JSR-347中加入了分布式代码执行与Map/reduce 的API 支持,各主流分布式缓存产品,如IBM WebSphere eXtreme Scale,VMware GemFire,GigaSpaces XAP 和Red Hat Infinispan 等也都支持这一新的编程模型.

1.2 典型应场景

分布式缓存的典型应用场景可分为以下几类:

1) 页面缓存.用来缓存Web 页面的内容片段,包括HTML、CSS 和图片等,多应用于社交网站等;

2) 应用对象缓存.缓存系统作为ORM 框架的二级缓存对外提供服务,目的是减轻数据库的负载压力,加速应用访问;

3) 状态缓存.缓存包括Session 会话状态及应用横向扩展时的状态数据等,这类数据一般是难以恢复的,对可用性要求较高,多应用于高可用集群;

4) 并行处理.通常涉及大量中间计算结果需要共享;

5) 事件处理.分布式缓存提供了针对事件流的连续查询(continuous query)处理技术,满足实时性需求;

6) 极限事务处理.分布式缓存为事务型应用提供高吞吐率、低延时的解决方案,支持高并发事务请求处理,多应用于铁路、金融服务和电信等领域.

# H2内存数据库

内存数据库抛弃了磁盘数据管理的传统方式，基于全部数据都在内存中重新设计了体系结构，并且在数据缓存、快速算法、并行操作方面也进行了相应的改进，所以数据处理速度比传统数据库的数据处理速度要快很多，一般都在10倍以上。内存数据库的最大特点是其“主拷贝”或“工作版本”常驻内存，即活动事务只与实时内存数据库的内存拷贝打交道。

定义:设有数据库系统DBS，DB为DBS中的数据库，DBM(t)为在时刻t，DB在内存的数据集，DBM(t)属于DB。TS为DBS中所有可能的事务构成的集合。AT(t)为在时刻t处于活动状态的事务集，AT(t)属于TS。Dt(T)为事务T在时刻t所操作的数据集，

Dt(T)属于DB。若在任意时刻t，均有:

任意T属于AT(t) Dt(T)属于DBM(t)

成立，则称DBS为一个内存数据库系统，简称为MMDBS;DB为一个内存数据库，简称为MMDB。

常见的例子有MySQL的MEMORY存储引擎、eXtremeDB、TT、FastDB、SQLite、Microsoft SQL Server Compact等

# 数据库连接池druid

阿里巴巴推出的国产数据库连接池，据网上测试对比，比目前的DBCP或C3P0数据库连接池性能更好

现在常用的开源数据库连接池主要有c3p0、dbcp、proxool三种，其中：

Spring 推荐使用dbcp；

Hibernate 推荐使用c3p0和proxool；

1、 DBCP：apache

DBCP(DataBase connection pool)数据库连接池。是apache上的一个 java连接池项目，也是 tomcat使用的连接池组件。单独使用dbcp需要3个包：common-dbcp.jar,common-pool.jar,common-collections.jar由于建立数据库连接是一个非常耗时耗资源的行为，所以通过连接池预先同数据库建立一些连接，放在内存中，应用程序需要建立数据库连接时直接到连接池中申请一个就行，用完后再放回去。dbcp没有自动的去回收空闲连接的功能。

2、 C3P0：

C3P0是一个开源的jdbc连接池，它实现了数据源和jndi绑定，支持jdbc3规范和jdbc2的标准扩展。c3p0是异步操作的，缓慢的jdbc操作通过帮助进程完成。扩展这些操作可以有效的提升性能。目前使用它的开源项目有Hibernate，Spring等。c3p0有自动回收空闲连接功能。

3、 Proxool：Sourceforge

Proxool是一种Java数据库连接池技术。是sourceforge下的一个开源项目,这个项目提供一个健壮、易用的连接池，最为关键的是这个连接池提供监控的功能，方便易用，便于发现连接泄漏的情况。

对比：

1> 相同时间内同等量的线程数和循环次数下:通过对三个连接池的三个标志性性能测试参数（Average,median,90%Line）进行比较发现：性能dbcp<=c3p0<proxool；

2> 不同情况下的同一数据库连接池测试：通过观察 Average,median,90%Line三个参数发

现三个连接池的稳定性（三种连接池的三个测试参数的变化情况）依次：稳定性dbcp>=c3p0>proxool。

结论：

通过对三种数据库连接池的性能测试发现，proxool和 c3p0能够更好的支持高并发，但是在稳定性方面略逊于 dpcp；

# 框架版本

Spring 4.2

Activiti 5.18

Mybatis 3.2

Jquery 1.81

# 集合框架

\* List:ArrayList LinkedList 元素可以重复

ArrayList：长于随机访问[插入和删除代价高] LinkedList长于插入和删除[随机访问慢] 可向上转型为Queue队列

Set:HashSet TreeSet LinkedHashSet 元素不能重复

HashSet无序[散列函数] 最快查找 TreeSet升序[红黑树]，LinkedHashSet按照插入顺序排序[散列,链表]

Map:HashMap TreeMap LinkedHashMap 键值对保存数据

HashMap 最快查找 TreeMap键值升序 LinkedHashMap按照插入顺序保存键,同时有HashMap查询速度

# 设计模式&单例&代理&命令

hh设计模式

AOP 分散关注，共享行为！

开闭原则，对扩展打开，对修改关闭。

高内聚，低耦合，迪米特原则

创建型模式

结构型模式

行为型模式

1 工厂方法模式

创建型模式，一个接口，几个实现类，一个接口工厂类，根据不同的参数创建不同的实例。但是增加子类需要修改工厂类。

2 抽象工厂模式

创建型模式，一个接口，几个实现类，多个接口工厂类，工厂类各自产生自己的实例，增加子类不需要修改工厂类，只需要扩展子类实现类和子类工厂类即可。做到了开闭原则。

3 单例模式

枚举创建单例模式线程安全的，一个jvm保证只有一个实例，对于一些大型的对象，这是一笔很大的系统开销，省去了new操作符，降低了系统内存的使用频率，减轻GC压力。有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。

单例模式和多例模式：有些系统层级的实例保证一个就够了，业务数据需要多个。

4 建造者模式（Builder）

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式的复数。

5 原型模式（Prototype）

从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。

深度复制～复制地址和引用互不影响。

浅度复制～复制引用，把原来对象改变

6 适配器模式

将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

7装饰模式和代理模式

相似的模式，拦截方法增加前后的逻辑

装饰模式就是给一个对象增加一些新的功能，而且是动态。

代理模式控制对象的访问，修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。静态代理和动态代理，静态代理增加方法必须修改代理类，动态代理不用，是实现AOP的灵魂，jdk支持的动态代理只支持接口实现类，cglib都支持，spring事务中接口代理用jdk，类代理用cglib.

8 外观模式和中介者模式

低耦合，是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中维护，只修改这个类的依赖关系。

9、享元模式（Flyweight）

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。通过连接池的管理，实现了数据库连接的共享，包括用户名，密码，连接窜等。不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能！

10 策略模式

父子类之间的行为模式，多个类的分别只是在于行为不同，共同继承一个父类，你需要对行为的算法做很多变动，客户不知道算法要使用的数据，设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数。

11 模板方法模式

父子类封装行为的模式，抽象父类中提供一个称之为模板方法的方法来定义这些基本方法的执行次序，而通过其子类来覆盖某些步骤，从而使得相同的算法框架可以有不同的执行结果。模板方法模式提供了一个模板方法来定义算法框架，而某些具体步骤的实现可以在其子类中完成。

Abstract A

doBusiness()

{

//接受参数

//校验

//保存逻辑

//保存相关联

//发邮件

//发消息

}

B C D

12观察者模式（Observer）

当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。发布方有一个集合装好订阅方，订阅方有一接口和一个实现类，通知方法迭代订阅方调用update方法。

13迭代子模式（Iterator）

迭代器模式就是顺序访问聚集中的对象。这句话包含两层意思：一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。jdk的迭代器就是用这个模式编写的。

14责任链模式（Chain of Responsibility）

有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求

h1.setHandler(h2);

h2.setHandler(h3);

15命令模式（Command）

命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！activiti采用命令模式执行数据库。发出者封装一个接受者的对象，执行里面的方法。

16备忘录模式（Memento）

主要目的是恢复一个对象的某个状态，假设有原始类A，A中有各种属性，A可以决定需要备份的属性，备忘录类B是用来存储A的一些内部状态，类C呢，就是一个用来存储备忘录的。用一个中间类保存初始状态，待修改后用中间类赋值。

17状态模式（State）

# 面向对象及面向接口编程

其实不论是面向接口还是面向对象还是面向方面编程，只是设计模式的一种体现，用最简单的话来解释就是：

面向对象编程就是一堆对象，通过封装、继承、多态等更加有效的组织程序。

面向接口编程就是一堆接口，通过接口规约对象的属性和方法，是面向对象一部分。

关于面向接口、面向对象、面向方面编程的比较区别\_simoniu\_新浪博客

面向方面编程把业务的主逻辑和次逻辑分开的一种思想。

面向对象不好解释，可以理解为以一切元素都是对象，在设计时以对象为单位，考虑它的属性及方法。设计中采用了封装、继承、抽象的手法

面向接口本身就是面向对象的，无所谓区别，只不过面向接口的好处是耦合性低

面向方面Aspect-Oriented Programming (AOP)就是大名鼎鼎的AOP。其实有点象struts里的拦截器。

举例：假设有在一个应用系统中，有一个共享的数据必须被并发同时访问，首先，将这个数据封装在数据对象中，称为Data Class，同时，将有多个访问类，专门用于在同一时刻访问这同一个数据对象。

为了完成上述并发访问同一资源的功能，需要引入锁Lock的概念，也就是说，某个时刻，当有一个访问类访问这个数据对象时，这个数据对象必须上锁Locked，用完后就立即解锁unLocked，再供其它访问类访问。 这种加锁与解锁的操作可以使用拦截器来实现。而拦截器本身就是面向方面编程的一个具体应用。

# Activiti和Drools

Activiti项目是一项新的基于Apache许可的开源BPM平台，从基础开始构建，旨在提供支持新的BPMN 2.0标准，包括支持对象管理组（OMG），面对新技术的机遇，诸如互操作性和云架构，提供技术实现。

创始人Tom Baeyens是JBoss jBPM的项目架构师，以及另一位架构师Joram Barrez，一起加入到创建Alfresco这项首次实现Apache开源许可的BPMN 2.0引擎开发中来。

Activiti是一个独立运作和经营的开源项目品牌，并将独立于Alfresco开源ECM系统运行。 Activiti将是一种轻量级，可嵌入的BPM引擎，而且还设计适用于可扩展的云架构。 Activiti将提供宽松的Apache许可2.0，以便这个项目可以广泛被使用，同时促进Activiti BPM引擎和的BPMN 2.0的匹配，该项目现正由OMG通过标准审定。 加入Alfresco Activiti项目的是VMware的SpringSource分支，Alfresco的计划把该项目提交给Apache基础架构，希望吸引更多方面的BPM专家和促进BPM的创新。

JBoss Rules 的前身是Codehaus的一个开源项目叫Drools。最近被纳入JBoss门下，更名为JBoss Rules，成为了JBoss[应用服务器](http://baike.baidu.com/view/995472.htm)的规则引擎。

Drools是为Java量身定制的基于Charles Forgy的[RETE算法](http://baike.baidu.com/view/4833633.htm)的规则引擎的实现。具有了OO接口的RETE,使得商业规则有了更自然的表达。

Drools的用XML的<Conditons>、<Consequence> 节点表达If--Then句式，而里面可以嵌入上述语言的代码作为判断语句和执行语句。

其中Java代码会使用Antlr进行解释，而Groovy和Python本身就是脚本语言，可以直接调用。

Drools的聪明之处在于，用XML节点来规范If--Then句式和事实的定义，使引擎干起活来很舒服。

而使用Java,Groovy等原生语言来做判断和执行语句，让程序员很容易过渡、移植，学习曲线很低。

1.      Deployment：流程部署对象，部署一个流程时创建。

2.      ProcessDefinitions：流程定义，部署成功后自动创建。

3.      ProcessInstances：流程实例，启动流程时创建。

4.      Task：任务，在Activiti中的Task仅指有角色参与的任务，即定义中的UserTask。

5.      Execution：执行计划，流程实例和流程执行中的所有节点都是Execution，如UserTask、ServiceTask等。

# 5.73 项目中遇到哪些代表性的困难，是怎么解决的？

1 AXIS接口超时

2 连接被重置

3 报表缓慢

4 activi异步生成代办

5 分布式事务锁等待

# 5.69 TB级大数据量和高并发处理方案

分区表

建立临时表

自动备份删除数据

分布式存储

分布式缓存

# 5.68 搜索引擎和全文索引

搜索引擎分类部分提到过全文搜索引擎从网站提取信息建立网页数据库的概念。搜索引擎的自动信息搜集功能分两种。一种是定期搜索，即每隔一

段时间（比如Google一般是28天），

蜘蛛搜索引擎搜索引擎主动派出“蜘蛛”程序，对一定IP地址范围内的互联网网站进行检索，一旦发现新的网站，它会自动提取网站的信息和网

址加入自己的数据库。

# 5.63 Mybatis的#和$比较

1. #将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #user\_id#，如果传入的值是111,那么解析成sql时的值为order by "111", 如果传入的值是id，则解析成的sql为order by "id".

2. $将传入的数据直接显示生成在sql中。如：order by $user\_id$，如果传入的值是111,那么解析成sql时的值为order by user\_id, 如果传入的值是id，则解析成的sql为order by id.

3. #方式能够很大程度防止sql注入。

4.$方式无法防止Sql注入。

5.$方式一般用于传入数据库对象，例如传入表名.

6.一般能用#的就别用$.

# 5.56 Dom 4j和DOM和SAX比较

1.dom:(Document Object Model,文档对象模型) W3C组织推荐的一种处理方式

sax:(Simple API for XML)不是官方标准，但它是XML社区事实上的标准，几乎所有的XML解析器都支持它。

Dom4j是一个简单、灵活的开放源代码的库。Dom4j是由早期开发JDOM的人分离出来而后独立开

发的。与JDOM不同的是，dom4j使用接口和抽象基类，虽然Dom4j的API相对要复杂一些，但它提

供了比JDOM更好的灵活性。

# 5.60 String,StringBuffer,StringBuilder比较

1.三者在执行速度方面的比较：StringBuilder > StringBuffer > String

我们明明就是改变了String型的变量s的，为什么说是没有改变呢? 其实这是一种欺骗，JVM是这样解析这段代码的：

首先创建对象s，赋予一个abcd，然后再创建一个新的对象s用来　　　　执行第二行代码，也就是说我们之前对象s

并没有变化，所以我们说String类型是不可改变的对象了，由于这种机制，每当用String操作字符串时，实际上是在

不断的创建新的对象，而原来的对象就会变为垃圾被ＧＣ回收掉，可想而知这样执行效率会有多底。

而StringBuffer与StringBuilder就不一样了，他们是字符串变量，是可改变的对象，每当我们用它们对字符串做操

作时，实际上是在一个对象上操作的，这样就不会像String一样创建一些而外的对象进行操作了，当然速度就快了。

StringBuilder：线程非安全的

StringBuffer：线程安全的

当我们在字符串缓冲去被多个线程使用是，JVM不能保证StringBuilder的操作是安全的，虽然他的速度最快，但是可

以保证StringBuffer是可以正确操作的。当然大多数情况下就是我们是在单线程下进行的操作，所以大多数情况下是建

议用StringBuilder而不用StringBuffer的，就是速度的原因。

# 5.53 Rest与WebService比较

REST的思想归结以下有如下几个关键点：

1．面向资源的接口设计轻量级

2．抽象操作为基础的CRUD

3．Http是应用协议而非传输协议

# 5.43 监听器和过滤器比较

1.过滤器：所谓过滤器顾名思义是用来过滤的，在java web中，你传入的request,response提前过滤掉一些信息，或者提前设置一些参数，然

后再传入servlet或者struts的action进行业务逻辑，比如过滤掉非法url（不是login.do的地址请求，如果用户没有登陆都过滤掉）,或者在

传入servlet或者struts的action前统一设置字符集，或者去除掉一些非法字符（聊天室经常用到的，一些骂人的话）。filter 流程是线性

的， url传来之后，检查之后，可保持原来的流程继续向下执行，被下一个filter, servlet接收等.

2.监听器：这个东西在c/s模式里面经常用到，他会对特定的事件产生产生一个处理。监听在很多模式下用到。比如说观察者模式，就是一个

监听来的。又比如struts可以用监听来启动。Servlet监听器用于监听一些重要事件的发生，监听器对象可以在事情发生前、发生后可以做一

些必要的处理。

3.java的拦截器 主要是用在插件上，扩展件上比如 hivernate spring struts2等 有点类似面向切片的技术，在用之前先要在配置文件即xml

文件里声明一段的那个东西。

# 5.16 单点登录

单点登录（Single Sign On），简称为 SSO，是目前比较流行的企业业务整合的解决方案之一。SSO的定义是在多个应用系统中，用户只需要登录

一次就可以访问所有相互信任的应用系统。它包括可以将这次主要的登录映射到其他应用中用于同一个用户的登录的机制。

业界已有很多产品支持SSO，如IBM的WebSphere和BEA的WebLogic，但各家SSO产品的实现方式也不尽相同。WebSphere通过Cookie记录认证信息，

WebLogic则是通过Session共享认证信息。Cookie是一种客户端机制，它存储的内容主要包括: 名字、值、过期时间、路径和域，路径与域合在一

起就构成了Cookie的作用范围，因此用Cookie方式可实现SSO，但域名必须相同; Session是一种服务器端机制，当客户端访问服务器时，服务器

为客户端创建一个惟一的SessionID，以使在整个交互过程中始终保持状态，而交互的信息则可由应用自行指定，因此用Session方式实现SSO，不

能在多个浏览器之间实现单点登录，但却可以跨域。

# Java编码解码字符集

1、在[Java](http://lib.csdn.net/base/java)开发中经常被文字乱码的问题困扰。下面全面解释下字符串的编码和解码。

如 String str = "中国"

编码：byte[] bts = str.getBytes("编码方式");//常用编码方式 gbk、utf-8、gb2312、iso-8859-1等等。

解码：String b = new String(bts,"解码方式");//解码方式对应常用编码方式。

2、常识：正常（正常显示的字符串无乱码）解码后的字符串对象可以用任意方式编码。但解码要正常显示，必须用对应的编码方式解码。（对于中文要保证正常显示必须采用中文编码/解码方式）

如 String str = "中国";//这个就是正常显示的字符串

以下任意方式（中文）编码/解码-----------

String b1 = new String(str.getBytes("gbk"),"gbk");

String b2 = new String(str.getBytes("utf-8"),"utf-8");

String b3 = new String(str.getBytes("gb2312"),"gb2312");

以上三种方式的字符串对象都是采用中文方式编码/解码，所以都会是正常并无乱码。

3、当没采用对应方式解码时（也就是所谓的乱码）怎么转成正常显示而无乱码。

如 String str = "中国";

String b1 =  new String(str.getBytes("gbk"),"gbk");//这里的b1就是采用的gbk的方式解码的 ：：：：b1不会乱码的

String b2 = new String(b1.getBytes(),"utf-8");//这里对b1编码后的字节数组重新用utf-8而没有用对应的gbk方式解码。：：：：：b2将会是乱码的

以下开始将乱码转成正常显示

String b3 = new String(b2.getBytes("utf-8"),"gbk")//这里编码方式（utf-8）必须用上述b2的解码方式,而解码方式（gbk）必须用b1的解码方式。因为是b2把正常显示的b1解码成乱码。：：：：：：：b3不会是乱码的。

在实际应用中出现的乱码，都是这个原因造成的，乱码不是不可逆的，上述步骤就是一个逆转的过程。

4常见应用举例。

在web开发中，tomcat对于传输的字符串都是采用iso-8859-1编码/解码方式。而客户端（浏览器端对于中文都是用gbk或utf-8中文编码/解码方式），所以传到后台都会是乱码的。容器一般都是有处理的，所以中文能正常显示和存储。但有些情况也是会出现乱码的，解决方式如下

String b = new String(str.getBytes("iso-8859-1","客户端的编码/解码方式")//中文解码方式一般用的是utf-8或者gbk。

如：

 String b1 = new String(str.getBytes("iso-8859-1","utf-8");

# SQL注入

参数化SQL；正则表达式；转义 SQL 语句中使用的字符串中的特殊字符；

# 软件开发流程

1、问题的定义及规划  
　　此阶段是软件开发与需求放共同讨论，主要确定软件的开发目标及其可行性。  
　　2、需求分析  
　　在确定软件开发可行性的情况下，对软件需要实现的各个功能进行详细需求分析。需求分析阶段是一个很重要的阶段，这一阶段做的好，将为整个软件项目的开发打下良好的基础。“唯一不变的是变化本身”，同样软件需求也是在软件爱你开发过程中不断变化和深入的，因此，必须定制需求变更计划来应付这种变化，以保护整个项目的正常进行。  
　　3、软件设计  
　　此阶段中偶要根据需求分析的结果，对整个软件系统进行设计，如系统框架设计、[数据库设计](http://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E5%BA%93%E8%AE%BE%E8%AE%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mHn3nAn4uH0LPj7WPH7h0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHmzrjnzrHT3)等。软件设计一般分为[总体设计](http://www.baidu.com/s?wd=%E6%80%BB%E4%BD%93%E8%AE%BE%E8%AE%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mHn3nAn4uH0LPj7WPH7h0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHmzrjnzrHT3)和详细设计。还的软件设计将为软件程序编写打下良好的基础。  
　　4、程序编码  
　　此阶段是将软件设计的结果转化为计算机可运行的程序代码。在程序编码中必定要制定统一、符合标准的编写规范。以保证程序的可读性、易维护性。提高程序的运行效率。  
　　5、[软件测试](http://www.baidu.com/s?wd=%E8%BD%AF%E4%BB%B6%E6%B5%8B%E8%AF%95&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mHn3nAn4uH0LPj7WPH7h0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHmzrjnzrHT3)  
　　在软件设计完成之后要进行严密的测试，一发现软件在整个软件设计过程中存在的问题并加以纠正。整个测试阶段分为[单元测试](http://www.baidu.com/s?wd=%E5%8D%95%E5%85%83%E6%B5%8B%E8%AF%95&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mHn3nAn4uH0LPj7WPH7h0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHmzrjnzrHT3)、组装测试、[系统测试](http://www.baidu.com/s?wd=%E7%B3%BB%E7%BB%9F%E6%B5%8B%E8%AF%95&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mHn3nAn4uH0LPj7WPH7h0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHmzrjnzrHT3)三个阶段进行。测试方法主要有白盒测试和黑盒测试IMG_256。

# 软件开发模式

快速原型模型：（需要迅速造一个可以运行的软件原型，以便理解和澄清问题）

快速原型模型允许在需求分析阶段对软件的需求进行初步的非完全的分析和定义，快速设计开发出软件系统的原型（展示待开发软件的全部或部分功能和性能

（过程：用户对该原型进行测试评定，给出具体改善的意见以及丰富的细化软件需求，开发人员进行修改完善）

优点：

克服瀑布模型的缺点，减少由于软件需求不明确带来的开发风险

缺点：

A、 所选用的开发技术和工具不一定符合主流的发展

B、 快速建立起来的系统加上连续的修改可能会造成 产品质量底下

增量模型：（采用随着日程时间的进展而交错的线性序列，每一个线性徐磊产生软件的一个可发布的“增量”，第一个增量往往就是核心的产品）

与其他模型共同之处：它与原型实现模型和其他演化方法一样，本质都是迭代

与原型实现模型不同之处：它强调每一个增量均发布一个可操作产品，（它不需要等到所有需求都出来，只要摸个需求的增量包出来即可进行开发）

优点：

1、 人员分配灵活，一开始不需要投入大量人力资源

2、 当配备人员不能在限定的时间内完成产品时，它可以提供一种先推出核心产品的途径，可现发布部分功能给用户（对用户起镇静作用）

3、 增量能够有计划的管理技术风险

缺点：

1、 如果增量包之间存在相交的情况且未很好处理，则必须做全盘系统分析

注：

这种模型将功能细化后分别开发的方法较适应于需求经常改变的软件开发过程

原型模型：（样品模型，采用逐步求精的方法完善原型）

主要思想：

先借用已有系统作为原型模型，通过“样品”不断改进，使得最后的产品就是用户所需要的。原型模型通过向用户提供原型获取用户的反馈，使开发出的软件能够真正反映用户的需求，

采用方法：

原型模型采用逐步求精的方法完善原型，使得原型能够“快速”开发，避免了像瀑布模型一样在冗长的开发过程中难以对用户的反馈作出快速的响应

优点：

(1）开发人员和用户在“原型”上达成一致。这样一来，可以减少设计中的错误和开发中的风险，也减少了对用户培训的时间，而提高了系统的实用、正确性以及用户的满意程度。

（2）缩短了开发周期，加快了工程进度。

（3）降低成本。

　缺点：

1、当重新生产该产品时，难以让用户接收，给工程继续开展带来不利因素。

　 2、不宜利用原型系统作为最终产品。采用原型模型开发系统，用户和开发者必须达成一致：

喷泉模型：（以用户需求为动力，以对象为驱动的模型，主要用于采用对象技术的软件开发项目）

它认为软件开发过程自下而上周期的各阶段是相互迭代和无间隙的特性

相互迭代：软件的摸个部分常常被重复工作多次，相关对象在每次迭代中随之加入渐进的软件成分

无间隙:它在各项活动之间没有明显边界（如分析和设计活动之间<由于对象概念的应用，表达分析，设计，实现等活动只用对象类和关系>）

优点：

1、 可以提高软件项目开发效率，节省开发时间，适应于面向对象的软件开发过程

不便之处：

1、由于喷泉模型在各个开发阶段是重叠的，因此在开发过程中需要大量的开发人员，因此不利于项目的管理。

2、这种模型要求严格管理文档，使得审核的难度加大，尤其是面对可能随时加入各种信息、需求与资料的情况

螺旋模型：（适合用于需求经常变化的项目<适合于大型复杂的系统>）

它主要是风险分析与评估，沿着螺线进行若干次迭代，

过程：

1、 制定计划：确定软件目标，选定实施方案，弄清项目开发的限制条件

2、 风险分析：分析评估所选方案，考虑如何识别和消除风险

3、 实施工程：实施软件开发和验证；

4、 客户评估：评价开发工作，提出修正建议，制定下一步计划。

优点：

1、 它由风险驱动，强调可选方案和约束条件从而支持软件的重用，有助于将软件质量作为特殊目标融入产品开发中

缺点：

1、 难以让用户确信这种烟花方法的结果是可以控制的

2、 建设周期长（而软件技术发展比较快，所以经常会出现软件开发完毕后，和当前的技术水平有很大的差距，无法满足当前用户的需求）

3、 除非软件开发人员擅长寻找可能的风险，准确的分析风险，否则将会带来更大的风险

瀑布模型：（从本质来讲，瀑布模型是一个软件开发架构,重复应用）

（核心思想：按工序将问题化简，将功能的实现与设计分开，便于分工协作，采用结构化的分析与设计方法将逻辑实现与物理实现分开，依照软件生命周期自上而下，相互衔接的次序<如同瀑布流水逐级下落>）

缺点：

1、 在项目各个阶段之间极少有反馈，各个阶段的划分完全固定，阶段之间产生大量的文档，增加了工作量

2、 用户只有在项目生命周期的后期才能看到结果，增加了开发的风险

3、 需要过多的强制完成日期和里程碑来跟踪各个项目的阶段

4、 在每个阶段都会产生循环反馈

（如果有信息未被覆盖或是发现问题了，必须返回到上一个阶段<甚至更前面的活动>并进行适当的修改,只有当上一阶段都被确认后才进行下一阶段）

5、 早期的错误可能要等到开发后期的测试阶段才能发现，进而带来严重的后果

优点：

1、 为项目提供了按阶段分的检查点

2、 当完成一个阶段后，只需要去关注后续阶段

3、 可在迭代模型中应用瀑布模型

按照瀑布模型的阶段划分，软件测试可以分为单元测试，集成测试，系统测试

注：由于每个阶段都会产生循环反馈，对于经常变化的项目而言，瀑布模型毫无价值，这种模型的线性过程太理想化，已不适合现代的软件开发模式

# Java安全技术

对于一般的开发人员来说，很少需要对安全领域内的基础技术进行深入的研究，但是鉴于日常系统开发中遇到的各种安全相关的问题，熟悉和了解这些安全技术的基本原理和使用场景还是非常必要的。本文将对非对称加密、数字摘要、数字签名、数字证书、SSL、HTTPS等这些安全领域内的技术进行一番简要的介绍，解释他们之间的关系，同时补充一些周边话题。

安全领域的技术众多，但是归根结底，他们都是为了保障如下三个方面：

1）认证用户和服务器，确保数据发送到正确的客户机和服务器

2）加密数据以防止数据中途被窃取

3）维护数据的完整性，确保数据在传输过程中不被改变。

对称加密和非对称密钥加解密

对于一份数据，通过一种算法，基于传入的密钥(一串由数字或字符组成的字符串,也称“key”），将明文数据转换成了不可阅读的密文，这是众所周知的“加密”，同样的，密文到达目的地后，需要再以相应的算法，配合一个密钥，将密文再解密成明文，这就是“解密”。如果加密和解密使用的是同一个密钥,那么这就是“对称密钥加解密”（最常见的对称加密算法是DES）。如果加密和解密使用的是两个不同的密钥，那么这就是“非对称密钥加解密”（最常用的非对称加密算法是RSA）。这两个不同的密钥一个叫作公开密钥（publickey）另一个叫私有密钥（privatekey），公开密钥对外公开，任何人均可获取，而私有密钥则由自己保存，其实公钥和私钥并没有什么不同之处，公钥之所以成为公钥是因为它会被公开出来，产生任意份拷贝，供任何人获取，而只有服务主机持有唯一的一份私钥，从这种分发模式上看，我们不难看出其中的用意，这种分发模式实际上是Web站点多客户端(浏览器)与单一服务器的网络拓扑所决定的，多客户端意味着密钥能被复制和公开获取，单一服务器意味着密钥被严格控制，只能由本服务器持有，这实际上也是后面要提到的之所以能通过数据证书确定信任主机的重要原因之一。如果我们跳出web站点的拓扑环境，其实就没有什么公钥与私钥之分了，比如，对于那些使用以密钥为身份认证的SSH主机，往往是为每一个用户单独生成一个私钥分发给他们自己保存，SSH主机会保存一份公钥，公钥私钥各有一份，都不会公开传播。

简言之：

对称加密速度快，但加密和解密的钥匙必须相同，只有通信双方才能知道钥匙

非对称加密速度慢，加密和解密的钥匙不相同，某一个人持有私钥，任何人都可以知道公钥

数字摘要--数据完整性的校验

这个非常简单，我们在下载文件的时候经常会看到有的下载站点也提供下载文件的“数字摘要“，供下载者验证下载后的文件是否完整，或者说是否和服务器上的文件”一模一样“。其实，数字摘要就是采用单项Hash函数将需要加密的明文“摘要”成一串固定长度（128位）的密文，这一串密文又称为数字指纹，它有固定的长度，而且不同的明文摘要成密文，其结果总是不同的，儿同样的明文其摘要必定一致。 因此，“数字摘要“叫”数字指纹“可能会更贴切一些。“数字摘要“是https能确保数据完整性和防篡改的根本原因。

数字签名--水到渠成的技术

让我们来看看有了“非对称密钥加解密”和“数字摘要“两项技术之后，我们能做些什么呢？假如发送方想把一份报文发送给接收方，在发送报文前，发送方用一个哈希函数从报文文本中生成报文摘要,然后用自己的私人密钥对这个摘要进行加密，这个加密后的摘要将作为报文的”签名“和报文一起发送给接收方，接收方首先用与发送方一样的哈希函数从接收到的原始报文中计算出报文摘要，接着再用发送方的公用密钥来对报文附加的数字签名进行解密，如果这两个摘要相同、那么接收方就能确认报文是从发送方发送且没有被遗漏和修改过！这就是结合“非对称密钥加解密”和“数字摘要“技术所能做的事情，这也就是人们所说的“数字签名”技术。在这个过程中，对传送数据生成摘要并使用私钥进行加密地过程就是生成”数字签名“的过程，经过加密的数字摘要，就是人们所说的”数字签名“！

数字签名技术就是对“非对称密钥加解密”和“数字摘要“两项技术的应用，它将摘要信息用发送者的私钥加密，与原文一起传送给接收者。接收者只有用发送者的公钥才能解密被加密的摘要信息，然后用HASH函数对收到的原文产生一个摘要信息，与解密的摘要信息对比。如果相同，则说明收到的信息是完整的，在传输过程中没有被修改，否则说明信息被修改过，因此数字签名能够验证信息的完整性。(注意，数字签名只能验证数据的完整性，数据本身是否加密不属于数字签名的控制范围)

综上所述，数字签名有两种功效：一是能确定消息确实是由发送方签名并发出来的，因为别人假冒不了发送方的签名。二是数字签名能确定消息的完整性。

数字证书--值得信赖的公钥

只从”准确认证发送方身份“和”确保数据完整性“两个安全方面来看，数字签名似乎已经完全做到了，还有漏洞存在的可能么？有，漏洞不在数字签名技术本身，而在它所依赖的密钥，只有密钥是真实可靠的前提下，使用数字签名才是安全有效的。考虑这种可能的情况：在上述发送方向接收方传送报文的例子中，如果发送方所持有的公钥来路有问题或是被替换了，那么，持有对应私钥的冒充接受方就有可能接收到发送方发送的报文。这里的问题就是：对于请求方来说，它怎么能确定它所得到的公钥一定是从目标主机那里发布的，而且没有被篡改过呢？亦或者请求的目标主机本本身就从事窃取用户信息的不正当行为呢？这时候，我们需要有一个权威的值得信赖的第三方机构(一般是由政府审核并授权的机构)来统一对外发放主机机构的公钥，只要请求方这种机构获取公钥，就避免了上述问题的发生。这种机构被称为证书权威机构（Certificate Authority, CA），它们所发放的包含主机机构名称、公钥在内的文件就是人们所说的“数字证书”。

数字证书的颁发过程一般为：用户首先产生自己的密钥对，并将公共密钥及部分个人身份信息传送给认证中心。认证中心在核实身份后，将执行一些必要的步骤，以确信请求确实由用户发送而来，然后，认证中心将发给用户一个数字证书，该证书内包含用户的个人信息和他的公钥信息，同时还附有认证中心的签名信息。用户就可以使用自己的数字证书进行相关的各种活动。数字证书由独立的证书发行机构发布。数字证书各不相同，每种证书可提供不同级别的可信度。可以从证书发行机构获得您自己的数字证书。（本段摘自百度百科）

SSL

SSL(Secure Socket Layer)是netscape公司设计的主要用于web的安全传输协议。这种协议在WEB上获得了广泛的应用，IETF(www.ietf.org)将SSL作了标准化，即RFC2246,并将其称为TLS（Transport Layer Security），从技术上讲，TLS1.0与SSL3.0的差别非常微小。

基本原理：先非对称加密传递对称加密所要用的钥匙，然后双方用该钥匙对称加密和解密往来的数据

要求：服务器端需安装数字证书，用户可能需要确认证书，会话过程中的加密与解密过程由浏览器与服务器自动完成，对用户完全透明。

SSL握手阶段示意图：

工作过程：

浏览器向服务器发出请求，询问对方支持的对称加密算法和非对称加密算法；服务器回应自己支持的算法。

浏览器选择双方都支持的加密算法，并请求服务器出示自己的证书；服务器回应自己的证书。

浏览器随机产生一个用于本次会话的对称加密的钥匙，并使用服务器证书中附带的公钥对该钥匙进行加密后传递给服务器；服务器为本次会话保持该对称加密的钥匙。第三方不知道服务器的私钥，即使截获了数据也无法解密。非对称加密让任何浏览器都可以与服务器进行加密会话。

浏览器使用对称加密的钥匙对请求消息加密后传送给服务器，服务器使用该对称加密的钥匙进行解密；服务器使用对称加密的钥匙对响应消息加密后传送给浏览器，浏览器使用该对称加密的钥匙进行解密。第三方不知道对称加密的钥匙，即使截获了数据也无法解密。对称加密提高了加密速度。

HTTPS

如果我们是在一开始来讲述HTTPS协议，那将会是一个很大的话题，但是讲到这里的时候，实现上所有关于HTTPS的内容，我们基本上已经讲完了，它所有依赖的所有安全技术就是上面我们所提及的，就像大家所知道的那样，HTTPS是由SSL+HTTP协议构建的可进行加密传输、身份认证（确认客户端连接的目标主机是否是真实正确的主机）的网络协议。https所能实现的安全保证，正是SSL所能解决的安全问题。

HTTPS的劣势：

https的主要缺点就是性能问题。造成https性能低于http的原因有两个：

1.对数据进行加解密决定了它比http慢。

2.另外一个重要原因的是https禁用了缓存。

相关测试数据表明使用HTTPS协议传输数据的工作效率只有使用HTTP协议传输的十分之一。因此对于一个网站来说，只有那对那些安全要求极高的的数据才会选择使用https进行传输。

对以上的知识联通起来做一个集中图示，相信大家会有更加清晰的理解：

1.鲍勃有两把钥匙，一把是公钥，另一把是私钥。

2.鲍勃把公钥送给他的朋友们----帕蒂、道格、苏珊----每人一把。

3.苏珊要给鲍勃写一封保密的信。她写完后用鲍勃的公钥加密，就可以达到保密的效果。

4.鲍勃收信后，用私钥解密，就看到了信件内容。这里要强调的是，只要鲍勃的私钥不泄露，这封信就是安全的，即使落在别人手里，也无法解密。

5.鲍勃给苏珊回信，决定采用"数字签名"。他写完后先用Hash函数，生成信件的数字摘要（digest）。

6.然后，鲍勃使用私钥，对这个数字摘要加密，生成"数字签名"（signature）。

7.鲍勃将这个签名，附在信件下面，一起发给苏珊。

8.苏珊收信后，取下数字签名，用鲍勃的公钥解密，得到信件的摘要。由此证明，这封信确实是鲍勃发出的。

9.苏珊再对信件本身使用Hash函数，将得到的结果，与上一步得到的摘要进行对比。如果两者一致，就证明这封信未被修改过。

10.复杂的情况出现了。道格想欺骗苏珊，他偷偷使用了苏珊的电脑，用自己的公钥换走了鲍勃的公钥。此时，苏珊实际拥有的是道格的公钥，但是还以为这是鲍勃的公钥。因此，道格就可以冒充鲍勃，用自己的私钥做成"数字签名"，写信给苏珊，让苏珊用假的鲍勃公钥进行解密。

11.后来，苏珊感觉不对劲，发现自己无法确定公钥是否真的属于鲍勃。她想到了一个办法，要求鲍勃去找"证书中心"（certificate authority，简称CA），为公钥做认证。证书中心用自己的私钥，对鲍勃的公钥和一些相关信息一起加密，生成"数字证书"（Digital Certificate）。

12.鲍勃拿到数字证书以后，就可以放心了。以后再给苏珊写信，只要在签名的同时，再附上数字证书就行了。

13.苏珊收信后，用CA的公钥解开数字证书，就可以拿到鲍勃真实的公钥了，然后就能证明"数字签名"是否真的是鲍勃签的。

14.我们看一个应用"数字证书"的实例：https协议。这个协议主要用于网页加密

15.首先，客户端向服务器发出加密请求。

16.服务器用自己的私钥加密网页以后，连同本身的数字证书，一起发送给客户端。

17.客户端（浏览器）的"证书管理器"，有"受信任的根证书颁发机构"列表。客户端会根据这张列表，查看解开数字证书的公钥是否在列表之内。

18.如果数字证书记载的网址，与你正在浏览的网址不一致，就说明这张证书可能被冒用，浏览器会发出警告。

19.如果这张数字证书不是由受信任的机构颁发的，浏览器会发出另一种警告。

20.如果数字证书是可靠的，客户端就可以使用证书中的服务器公钥，对信息进行加密，然后与服务器交换加密信息。

# JVM性能调优



1. JVM性能调优有很多设置,这个参考JVM参数即可.
2. 主要调优的目的:
3. 控制GC的行为.GC是一个后台处理,但是它也是会消耗系统性能的,因此经常会根据系统运行的程序的特性来更改GC行为
4. 控制JVM堆栈大小.一般来说,JVM在内存分配上不需要你修改,(举例)但是当你的程序新生代对象在某个时间段产生的比较多的时候,就需要控制新生代的堆大小.同时,还要需要控制总的JVM大小避免内存溢出
5. 控制JVM线程的内存分配.如果是多线程程序,产生线程和线程运行所消耗的内存也是可以控制的,需要通过一定时间的观测后,配置最优结果

一、GC概要

JVM堆相关知识

为什么先说JVM堆?

JVM的堆是Java对象的活动空间，程序中的类的对象从中分配空间，其存储着正在运行着的应用程序用到的所有对象。这些对象的建立方式就是那些new一类的操作，当对象无用后，是GC来负责这个无用的对象(地球人都知道)。

JVM堆

(1) 新域：存储所有新成生的对象

(2) 旧域：新域中的对象，经过了一定次数的GC循环后，被移入旧域

(3)永久域：存储类和方法对象，从配置的角度看，这个域是独立的，不包括在JVM堆内。默认为4M。

新域会被分为3个部分：1.第一个部分叫Eden。(伊甸园？？可能是因为亚当和夏娃是人类最早的活动对象？)2.另两个部分称为辅助生存空间(幼儿园)，我这里一个称为A空间(From sqace)，一个称为B空间(To Space)。

二、GC浅谈

GC的工作目的很明确：在堆中，找到已经无用的对象，并把这些对象占用的空间收回使其可以重新利用.大多数垃圾回收的 算法思路都是一致的：把所有对象组成一个集合，或可以理解为树状结构，从树根开始找，只要可以找到的都是活动对象，如果找不到，这个对象就是凋零的昨日黄 花，应该被回收了。

在sun 的文档说明中，对JVM堆的新域，是采用coping算法，该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收。它开始时把堆分成一个对象面和多个 空闲面，程序从对象面为对象分配空间，当对象满了，基于 coping算法的垃圾收集就从根集中扫描活动对象，并将每个活动对象复制到空闲面(使得活动对象所占的内存之间没有空闲洞)，这样空闲面变成了对象面， 原来的对象面变成了空闲面，程序会在新的对象面中分配内存。

对于新生成的对象，都放在Eden中；当Eden充满时（小孩太多 了），GC将开始工作，首先停止应用程序的运行，开始收集垃圾，把所有可找到的对象都复制到A空间中，一旦当A空间充满，GC就把在A空间中可找到的对象 都复制到B空间中(会覆盖原有的存储对象)，当B空间满的时间，GC就把在B空间中可找到的对象都复制到A空间中，AB在这个过程中互换角色，那位客官说 了：拷来拷去，烦不烦啊？什么时候是头？您别急，在活动对象经过一定次数的GC操作后，这些活动对象就会被放到旧域中。对于这些活动对象，新域的幼儿园生 活结束了。新域为什么要这么折腾？起初在这块我也很迷糊，又查了些资料，原来是这样：应用程序生成的绝大部分对象都是短命的，copying算法最理想的 状态是，所有移出Eden的对象都会被收集，因为这些都是短命鬼，经过一定次数的GC后应该被收集，那么移入到旧域的对象都是长命的，这样可以防止AB空 间的来回复制影响应用程序。实际上这种理想状态是很难达到的，应用程序中不可避免地存在长命的对象，copying算法的发明者要这些对象都尽量放在新域 中，以保证小范围的复制，压缩旧域的开销可比新域中的复制大得多(旧域在下面说)。对于旧域，采用的是tracing算法的一种，称为标记-清除-压缩收 集器，注意，这有一个压缩，这是个开销挺大的操作。垃圾回收主要是对Young Generation块和Old Generation块内存进行回收，YG用来放新产生的对象，经过几次回收还没回收掉的对象往OG中移动，对YG进行垃圾回收又叫做MinorGC，对 OG垃圾回收又叫MajorGC，两块内存回收互不干涉。二、Gc 流程：

[older generation][survivor 1][survivor 2][eden]

\*young generation=eden + survivor

1.当eden满了，触发young GC；

2.young GC做2件事：一，去掉一部分没用的object；二，把老的还被引用的object发到survior里面，等下几次GC以后，survivor再放到old里面。

3.当old满了，触发full GC。full GC很消耗内存，把old，young里面大部分垃圾回收掉。这个时候用户线程都会被block。

三、young generation比例越大,不一定最好。

将young的大小设置为大于总堆大小的一半时会造成效率低下。如果设置得过小，又会因为young generation收集程序不得不频繁运行而造成瓶颈。

四、总结

从上面的推导可以得出很多结论，下面是前辈的经验总结与自已的认识

1.JVM堆的大小决定了GC的运行时间。如果JVM堆的大小超过一定的限度，那么GC的运行时间会很长。

2.对象生存的时间越长，GC需要的回收时间也越长，影响了回收速度。

3.大多数对象都是短命的，所以，如果能让这些对象的生存期在GC的一次运行周期内，wonderful！

4.应用程序中，建立与释放对象的速度决定了垃圾收集的频率。

5.如果GC一次运行周期超过3-5秒，这会很影响应用程序的运行，如果可以，应该减少JVM堆的大小了。

6.前辈经验之谈：通常情况下，JVM堆的大小应为物理内存的80%。

# 源码专题

## Spring AOP

## Spring IOC,Bean

**上面的4个类封装了Mapper接口动态生成代理类的全部细节**

**MapperRegistry 类是注册Mapper接口与获取代理类实例的工具类**

[IMG_265](http://blog.csdn.net/jjyy778899/article/details/javascript:;)

//一个内部类 封装了具体执行的动作

  public static class SqlCommand {

    //xml标签的id

    private final String name;

    //insert update delete select的具体类型

    private final SqlCommandType type;

    public SqlCommand(Configuration configuration, Class<?> mapperInterface, Method method) throws BindingException {

      //拿到全名 比如 org.mybatis.example.BlogMapper.selectBlog

      String statementName = mapperInterface.getName() + "." + method.getName();

      MappedStatement ms = null;

      //获取MappedStatement对象 这个对象封装了XML当中一个标签的所有信息 比如下面

      //<select id="selectBlog" resultType="Blog">

      //select \* from Blog where id = #{id}

      //</select>

      if (configuration.hasStatement(statementName)) {

        ms = configuration.getMappedStatement(statementName);

      } else if (!mapperInterface.equals(method.getDeclaringClass().getName())) { // 这里是一个BUG

        String parentStatementName = method.getDeclaringClass().getName() + "." + method.getName();

        if (configuration.hasStatement(parentStatementName)) {

          ms = configuration.getMappedStatement(parentStatementName);

        }

      }

      //为空抛出异常

      if (ms == null) {

        throw new BindingException("Invalid bound statement (not found): " + statementName);

      }

      name = ms.getId();

      type = ms.getSqlCommandType();

      //判断SQL标签类型 未知就抛异常

      if (type == SqlCommandType.UNKNOWN) {

        throw new BindingException("Unknown execution method for: " + name);

      }

    }

    public String getName() {

      return name;

    }

    public SqlCommandType getType() {

      return type;

    }

  }

  //内部类 封装了接口当中方法的 参数类型 返回值类型 等信息

  public static class MethodSignature {

    //是否返回多调结果

    private final boolean returnsMany;

    //返回值是否是MAP

    private final boolean returnsMap;

    //返回值是否是VOID

    private final boolean returnsVoid;

    //返回值类型

    private final Class<?> returnType;

    //mapKey

    private final String mapKey;

    //resultHandler类型参数的位置

    private final Integer resultHandlerIndex;

    //rowBound类型参数的位置

    private final Integer rowBoundsIndex;

    //用来存放参数信息

    private final SortedMap<Integer, String> params;

    //是否存在命名参数

    private final boolean hasNamedParameters;

    //在这里对上面的属性进行初始化 就不一一详细说明了 具体实现细节可以看下面的代码。

    public MethodSignature(Configuration configuration, Method method) throws BindingException {

      this.returnType = method.getReturnType();

      this.returnsVoid = void.class.equals(this.returnType);

      this.returnsMany = (configuration.getObjectFactory().isCollection(this.returnType) || this.returnType.isArray());

      this.mapKey = getMapKey(method);

      this.returnsMap = (this.mapKey != null);

      this.hasNamedParameters = hasNamedParams(method);

      this.rowBoundsIndex = getUniqueParamIndex(method, RowBounds.class);

      this.resultHandlerIndex = getUniqueParamIndex(method, ResultHandler.class);

      this.params = Collections.unmodifiableSortedMap(getParams(method, this.hasNamedParameters));

    }

    public Object convertArgsToSqlCommandParam(Object[] args) {

      final int paramCount = params.size();

      if (args == null || paramCount == 0) {

        return null;

      } else if (!hasNamedParameters && paramCount == 1) {

        return args[params.keySet().iterator().next()];

      } else {

        final Map<String, Object> param = new ParamMap<Object>();

        int i = 0;

        for (Map.Entry<Integer, String> entry : params.entrySet()) {

          param.put(entry.getValue(), args[entry.getKey()]);

          // issue #71, add param names as param1, param2...but ensure backward compatibility

          final String genericParamName = "param" + String.valueOf(i + 1);

          if (!param.containsKey(genericParamName)) {

            param.put(genericParamName, args[entry.getKey()]);

          }

          i++;

        }

        return param;

      }

    }

    public boolean hasRowBounds() {

      return (rowBoundsIndex != null);

    }

    public RowBounds extractRowBounds(Object[] args) {

      return (hasRowBounds() ? (RowBounds) args[rowBoundsIndex] : null);

    }

    public boolean hasResultHandler() {

      return (resultHandlerIndex != null);

    }

    public ResultHandler extractResultHandler(Object[] args) {

      return (hasResultHandler() ? (ResultHandler) args[resultHandlerIndex] : null);

    }

    public String getMapKey() {

      return mapKey;

    }

    public Class<?> getReturnType() {

      return returnType;

    }

    public boolean returnsMany() {

      return returnsMany;

    }

    public boolean returnsMap() {

      return returnsMap;

    }

    public boolean returnsVoid() {

      return returnsVoid;

    }

    private Integer getUniqueParamIndex(Method method, Class<?> paramType) {

      Integer index = null;

      final Class<?>[] argTypes = method.getParameterTypes();

      for (int i = 0; i < argTypes.length; i++) {

        if (paramType.isAssignableFrom(argTypes[i])) {

          if (index == null) {

            index = i;

          } else {

            throw new BindingException(method.getName() + " cannot have multiple " + paramType.getSimpleName() + " parameters");

          }

        }

      }

      return index;

    }

    private String getMapKey(Method method) {

      String mapKey = null;

      if (Map.class.isAssignableFrom(method.getReturnType())) {

        final MapKey mapKeyAnnotation = method.getAnnotation(MapKey.class);

        if (mapKeyAnnotation != null) {

          mapKey = mapKeyAnnotation.value();

        }

      }

      return mapKey;

    }

    private SortedMap<Integer, String> getParams(Method method, boolean hasNamedParameters) {

      final SortedMap<Integer, String> params = new TreeMap<Integer, String>();

      final Class<?>[] argTypes = method.getParameterTypes();

      for (int i = 0; i < argTypes.length; i++) {

        if (!RowBounds.class.isAssignableFrom(argTypes[i]) && !ResultHandler.class.isAssignableFrom(argTypes[i])) {

          String paramName = String.valueOf(params.size());

          if (hasNamedParameters) {

            paramName = getParamNameFromAnnotation(method, i, paramName);

          }

          params.put(i, paramName);

        }

      }

      return params;

    }

    private String getParamNameFromAnnotation(Method method, int i, String paramName) {

      final Object[] paramAnnos = method.getParameterAnnotations()[i];

      for (Object paramAnno : paramAnnos) {

        if (paramAnno instanceof Param) {

          paramName = ((Param) paramAnno).value();

        }

      }

      return paramName;

    }

    private boolean hasNamedParams(Method method) {

      boolean hasNamedParams = false;

      final Object[][] paramAnnos = method.getParameterAnnotations();

      for (Object[] paramAnno : paramAnnos) {

        for (Object aParamAnno : paramAnno) {

          if (aParamAnno instanceof Param) {

            hasNamedParams = true;

            break;

          }

        }

      }

      return hasNamedParams;

    }

  }

}

MapperMethod

**通过上面的分析就很容易弄清楚Mybatis是如何利用JDK动态代理的机制生成代理类来对各种Mapper接口进行封装的了。**

## Jquery

## Requirejs

## string 源码

[Java](http://lib.csdn.net/base/java)中的String类可谓是重中之重，java保证其不变性，其有很多方法例如substring() , concat() , replace()  , join(), toLowerCase() , trim() , format() 等都似乎是会改变现有的对象，当然这不会发生。读String类的源代码就可以知道，一般是new String()方法，这些方法实际上返回的是一个新的String对象而并没有改变原来的String对象。

String类中有两个很重要的属性，其一为private final char value[];，其二为private int hash; 。hash为String对象的哈希值，这是为了缓存该对象的hash码，当再一次使用的时候不用再次花费时间计算，这点可以在其hashCode()方法的源代码中看到：  
public int hashCode() {  
        int h = hash;  
        if (h == 0 && value.length > 0) {  
            char val[] = value;  
            for (int i = 0; i < value.length; i++) {  
                h = 31 \* h + val[i];  
            }  
            hash = h;  
        }  
        return h;  
    }

另外value字符串数组为final的，用于保存String对象中的字符。属性就这么简单。之后看一些String类的方法：

首先构造方法就有很多，目的就是为了提供多种创建String对象的途径，例如空参数的public String()他知识new了一个空的char数组赋给value；public String(String original)通过直接赋值来创新一模一样的String对象；接受char数组的public String(char value[])通过Arrays.copyOf()方法来创建，而追根溯源之后发现Arrays.copyOf()方法重载了很多，当然参数不一样，是为了可以复制不同种类型的对象，其中Arrays.copyOf(char [] original , int newLength)使用System.arraycopy()方法来进行复制，而System.arraycopy()方法则是native的无法看到源码，令人印象深刻的是10个copyOf()方法中有8个用了System.arraycopy()来实现内部运作。在java中实现复制可以有多重方法，例如循环，或者使用clone()方法，或者使用效率最高的System.arraycopy()方法。然而这个System.arraycopy()方法在java类中随处可见，例如java.util包中很多容器类在使用，例如ArrayList类（jdk源代码jdk 1.8）：

public boolean addAll(int index, Collection<? extends E> c) {  
        rangeCheckForAdd(index);  
        Object[] a = c.toArray();  
        int numNew = a.length;  
        ensureCapacityInternal(size + numNew);  // Increments modCount  
  
        int numMoved = size - index;  
        if (numMoved > 0)  
            System.arraycopy(elementData, index, elementData, index + numNew,  
                             numMoved);  
  
        System.arraycopy(a, 0, elementData, index, numNew);  
        size += numNew;  
        return numNew != 0;  
    }

 protected void removeRange(int fromIndex, int toIndex) {  
        modCount++;  
        int numMoved = size - toIndex;  
        System.arraycopy(elementData, toIndex, elementData, fromIndex,  
                         numMoved);  
  
        // clear to let GC do its work  
        int newSize = size - (toIndex-fromIndex);  
        for (int i = newSize; i < size; i++) {  
            elementData[i] = null;  
        }  
        size = newSize;  
    }

String类中的getChars()方法：

public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin) {  
        if (srcBegin < 0) {  
            throw new StringIndexOutOfBoundsException(srcBegin);  
        }  
        if (srcEnd > value.length) {  
            throw new StringIndexOutOfBoundsException(srcEnd);  
        }  
        if (srcBegin > srcEnd) {  
            throw new StringIndexOutOfBoundsException(srcEnd - srcBegin);  
        }  
        System.arraycopy(value, srcBegin, dst, dstBegin, srcEnd - srcBegin);  
    }

还是回到String类中。重点应该看看equals()方法：

 public boolean equals(Object anObject) {  
        if (this == anObject) {  
            return true;  
        } //先进行同引用判断，equals方法中的套路  
        if (anObject  instanceof  String) {  
            String anotherString = (String)anObject; //在判断传参是String类型之后强制类型转换  
            int n = value.length;  
            if (n == anotherString.value.length) {  
                char v1[] = value;  
                char v2[] = anotherString.value;  
                int i = 0;  
                while (n-- != 0) {  
                    if (v1[i] != v2[i])  
                        return false;  
                    i++;  
                } //String类也是一个字符一个字符进行比较来判断两方是否equals  
                return true;  
            }  
        }  
        return false;  
    }

String类的compareTo()方法：

public int compareTo(String anotherString) {  
        int len1 = value.length;  
        int len2 = anotherString.value.length;  
        int lim = Math.min(len1, len2);  
        char v1[] = value;  
        char v2[] = anotherString.value;  
  
        int k = 0;  
        while (k < lim) {  
            char c1 = v1[k];  
            char c2 = v2[k];  
            if (c1 != c2) {  
                return c1 - c2; //常规的逐个字符比较  
            }  
            k++;  
        }  
        return len1 - len2;  
    }

总结：String类中的方法常用而且使用，[算法](http://lib.csdn.net/base/datastructure)并不是惊天级别的。所以好好看源代码是很有好处的。

# HTTPS理解

HTTPS和HTTP协议相比提供了

1. 数据完整性：内容传输经过完整性校验

2. 数据隐私性：内容经过对称加密，每个连接生成一个唯一的加密密钥

3. 身份认证：第三方无法伪造服务端（客户端）身份

[明文] 客户端发送随机数client\_random和支持的加密方式列表

2. [明文] 服务器返回随机数server\_random，选择的加密方式和服务器证书链

3. [RSA] 客户端验证服务器证书，使用证书中的公钥加密premaster secret发送给服务端

4. 服务端使用私钥解密premaster secret

5. 两端分别通过client\_random，server\_random和premaster secret生成master secret，用于对称加密后续通信内容

证书中包含什么信息：

• 证书信息：过期时间和序列号

• 所有者信息：姓名等

• 所有者公钥

为什么服务端要发送证书给客户端？

互联网有太多的服务需要使用证书来验证身份，以至于客户端（操作系统或浏览器等）无法内置所有证书，需要通过服务端将证书发送给客户端。

客户端为什么要验证接收到的证书？

中间人攻击

# 5.9 集群和负载均衡

1. 集群(Cluster)：是一组独立的计算机系统构成一个松耦合的多处理器系统，它们之间通过网络实现进程间的通信。应用程序可以通过网络共享内存

进行消息传送，实现分布式计算机。

2. 负载均衡(Load Balance)：先得从集群讲起，集群就是一组连在一起的计算机，从外部看它是一个系统，各节点可以是不同的操作系统或不同硬件

构成的计算机。如一个提供Web服务的集群，对外界来看是一个大Web服务器。不过集群的节点也可以单独提供服务。

3. 特点：在现有网络结构之上，负载均衡提供了一种廉价有效的方法扩展服务器带宽和增加吞吐量，加强网络数据处理能力，提高网络的灵活性和可

用性。集群系统 (Cluster)主要解决下面几个问题：高可靠性（HA）：利用集群管理软件，当主服务器故障时，备份服务器能够自动接管主服务器

的工作，并及时切换过去，以实现对用户的不间断服务。高性能计算（HP）：即充分利用集群中的每一台计算机的资源，实现复杂运算的并行处

理，通常用于科学计算领域，比如基因分析，化学分析等。负载平衡：即把负载压力根据某种算法合理分配到集群中的每一台计算机上，以减轻

主服务器的压力，降低对主服务器的硬件和软件要求。

# 33 NOSQL

NoSQL，是not only sql，是非关系数据库，不同于oracle等关系数据库。hadoop,是分布式解决方案，即为Mapreduce（计算的）

和HDFS（文件系统）,使用Hadoop和NoSQL可以构造海量数据解决方案。

NoSQL：是一项全新的数据库革命性运动，NoSQL的拥护者们提倡运用非关系型的数据存储。现今的计算机体系结构在数据存储方面要求具

备庞大的水平扩 展性，而NoSQL致力于改变这一现状。

是NoSQL数据库之间的不同，远超过两 SQL数据库之间的差别。这意味着软件架构师更应该在项目开始时就选择好一个适合的 NoSQL数据库

。针对这种情况，这里对 Cassandra、Mongodb、CouchDB、Redis、 Riak、Membase、Neo4j 和 HBase

# 5.4 UML

Unified Modeling Language (UML)又称统一建模语言或标准建模语言，是始于1997年一个OMG标准，它是一个支持模型化和软件系统开

发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持，包括由需求分析到规格，到构造和配置。 面向对象的分析与设计

(OOA&D，OOAD)方法的发展在80年代末至90年代中出现了一个高潮，UML是这个高潮的产物。它不仅统一了Booch、Rumbaugh和Jacobson的

表示方法，而且对其作了进一步的发展，并最终统一为大众所接受的标准建模语言。

事物是对模型中最具有代表性的成分的抽象，包括结构事物，如类（Class）、接口（Interface）、协作（Collaboration）、用例（UseCase）

、主动类（ActiveClass）、组件（Component）和节点（Node）；行为事物，如交互（Interaction）、态机（Statemachine）、分组事物

（包，Package）、注释事物（注解，Note）。

# 2.22 报表组件Jasperreports

Jasperreports和**HighChat**

Highcharts是一个非常流行，界面美观的纯Javascript图表库。它主要包括两个部分：Highcharts和Highstock。

Highcharts可以为您的网站或Web应用程序提供直观，互动式的图表。目前支持线，样条，面积，areaspline，柱形图，条形图，饼图和散点图类型。

Highstock可以为您方便地建立股票或一般的时间轴图表。它包括先进的导航选项，预设的日期范围，日期选择器，滚动和平移等等。

如果想要了解更多Highcharts的信息

# 15 项目管理五大因素

资源、成本、功能、质量、进度

# 2.1 Servlet和Jsp

1.jsp经编译后就变成了Servlet.(JSP的本质就是Servlet，JVM只能识别java的类，不能识别JSP的代码,Web容器将JSP的代码编译成JVM能够识别的java类)

2.jsp更擅长表现于页面显示,servlet更擅长于逻辑控制.

3.Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到.

Jsp是Servlet的一种简化，使用Jsp只需要完成程序员需要输出到客户端的内容，Jsp中的Java脚本如何镶嵌到一个类中，由Jsp容器完成。而Servlet则是个完整的Java类，这个类的Service方法用于生成对客户端的响应。

# 13 敏捷开发

敏捷开发以用户的需求进化为核心，采用迭代、循序渐进的方法进行软件开发。在敏捷开发中，软件项目在构建初期被切分成多个子项目，

各个子项目的成果都经过测试，具备可视、可集成和可运行使用的特征。换言之，就是把一个大项目分为多个相互联系，但也可独立运行的

小项目，并分别完成，在此过程中软件一直处于可使用状态。

快速适应现实的变化,迭代周期更短，更灵活

瀑布模式严格遵循预先计划的需求、分析、设计、编码、测试的步骤顺序进行。

# 13 MQ

MQ传递主干， 在世界屡获殊荣。 它帮您搭建企业服务总线(ESB)的基础传输层。IBM WebSphere MQ为SOA提供可靠的消息传递。它为经过验证的

消息传递主干， 全方位、 多用途的数据传输， 并帮助您搭建企业服务总线的传输基础设施。

消息队列（MQ）是一种应用程序对应用程序的通信方法。应用程序通过写和检索出入列队的针对应用程序的数据（消息）来通信，而无需专用连

接来链接它们。消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过直接调用彼此来通信，直接调用通常是用于诸如远程过程调

用的技术。排队指的是应用程序通过队列来通信。队列的使用除去了接收和发送应用程序同时执行的要求。

# 9 ETL

mapping,Resource,Target,WorkFlow,

# 8 Jquery/EXT/JAVASCRIPT/css

jQuery是一个快速的，简洁的javaScript库，使用户能更方便地处理HTML documents、events、实现动画效果，并且方便地为网站提供AJAX交互。

jQuery还有一个比较大的优势是，它的文档说明很全，而且各种应用也说得很详细，同时还有许多成熟的插件可供选择。

.EXT体积大

EXTJS和其他JS框架对比

1 轻量级的选择：

主要是mootools和jquery，由于它们的设计思想的不同，jQuery是追求简洁和高效，Mootools除了追求这些目标以外，其核心在于面向对象，所以jQuery适合于快速开发，Mootools适合于稍大型和复杂的项目，

其中需要面向对象的支持；另外，在Ajax的支持上，jQuery稍强一些；在Comet的支持上，jQuery有相关的插件，Mootools目前没有，但是Comet的核心在于服务器的支持，浏览器端的接口很简单，开发相关的插件很简单。

在面向对象的Javascript Library中，mootools逐渐战胜了prototype（体积大，面向对象的设计不合理等），也包括script.acul.ous（基于prototype，实际上就是prototype上的UI库）。

2 面向RIA的框架

考虑纯JavaScripty库，目前主要是Dojo和ExtJS（还有YUI）。Dojo更适合企业应用和产品开发的需要，因为离线存储、DataGrid、2D、3D图形、Chart、Comet等组件对于企业应用来说都是很重要的（当然这些组件

还要等一段时间才能稳定下来）。例如，BEA基于Mashup技术开发的产品中已经使用了Dojo。

ExtJS：美观和"易用"，并且足够强大。在对UI有比较大的需求时，是首选。

Cas sessionId

A B C

# 7 Struts1和Struts2

Action 类:

• Struts1 要求Action类继承一个抽象基类。Struts1 的一个普遍问题是使用抽象类编程而不是接口。

• Struts 2 Action类可以实现一个Action接口，也可实现其他接口，使可选和定制的服务成为可能。Struts2 提供一个ActionSupport基类去 实现 常用的接口。Action接口不是必须的，任何有execute标识的POJO对象都可以用作Struts2 的Action对象。

线程模式:

• Struts1 Action是单例模式并且必须是线程安全的，因为仅有Action的一个实例来处理所有的请求。单例策略限制了Struts1 Action能作的事，并且要在开发时特别小心。Action资源必须是线程安全的或同步的。

• Struts2 Action对象为每一个请求产生一个实例，因此没有线程安全问题。（实际上，servlet容器给每个请求产生许多可丢弃的对象，并且不会导致性能和垃圾回收问题）

Servlet 依赖:

• Struts1 Action 依赖于Servlet API ,因为当一个Action被调用时HttpServletRequest 和 HttpServletResponse 被传递给execute方法。

• Struts 2 Action不依赖于容器，允许Action脱离容器单独被测试。如果需要，Struts2 Action仍然可以访问初始的request和response。但是，其他的元素减少或者消除了直接访问HttpServetRequest 和 HttpServletResponse的必要性。

可测性:

• 测试Struts1 Action的一个主要问题是execute方法暴露了servlet API（这使得测试要依赖于容器）。一个第三方扩展－－Struts TestCase－－提供了一套Struts1 的模拟对象（来进行测试）。

• Struts 2 Action可以通过初始化、设置属性、调用方法来测试，“依赖注入”支持也使测试更容易。

# 1 Jboss/TOMCAT/WEBSPHERE

JBoss 是一个管理 EJB 的容器和服务器，支持 EJB 1.1、EJB 2.0 和 EJB3.0 的规范。但 JBoss 核 心服务不包括支持 servlet/JSP 的 WEB 容器，

一般与 Tomcat 或 Jetty 绑定使用。被 Redhat 公司收购.免费的。

weblogic WebLogic 是美国 bea 公司出品的一个 application server 确切的说是一个基于 j2ee 架构的中 间件。BEA WebLogic 是用于

开发、集成、部署和管理大型分布式 Web 应用、网络应用和数 据库应用的 Java 应用服务器。速度最快的服务器。

Tomcat支持JSP/SERVLET,不支持EJB。

# 4 Xfire,CXF和Axis

Axis2的最新版本是1.3.CXF现在已经到了2.0版本。CXF则是XFire和Celtix项目的结合产品,

Axis2支持多语言-除了Java,他还支持C/C++版本。

CXF可以很好支持Spring。Axis2不能

# 5 Spring BOOT

1：创建独立的spring应用。

2：嵌入Tomcat, Jetty Undertow 而且不需要部署他们。

3：提供的“starters”poms来简化Maven配置

4：尽可能自动配置spring应用。

5：提供生产指标,健壮检查和外部化配置

6：绝对没有代码生成和XML配置要求

# 6 Mybatis和Hibernate

则在于POJO 与SQL之间的映射关系。然后通过映射配置文件，将SQL所需的参数，以及返回的结果字段映射到指定POJO。相对Hibernate“O/R”而言，iBATIS

是一种“Sql Mapping”的ORM实现。

Mybatis框架相对简单很容易上手,Mybatis需要手动编写SQL语句，以及ResultMap。而Hibernate有良好的映射机制，开发者无需关心SQL的生成与结果映射，

可以更专注于业务流程。

Hibernate与具体数据库的关联只需在XML文件中配置即可，所有的HQL语句与具体使用的数据库无关，移植性很好。MyBatis项目中所有的SQL语句都是依赖所

用的数据库的，所以不同数据库类型的支持不好

Hibernate缓存

Hibernate一级缓存是Session缓存，利用好一级缓存就需要对Session的生命周期进行管理好。建议在一个Action操作中使用一个Session。一级缓存需要对

Session进行严格管理。

Hibernate二级缓存是SessionFactory级的缓存。 SessionFactory的缓存分为内置缓存和外置缓存。内置缓存中存放的是SessionFactory对象的一些集合属

性包含的数据(映射元素据及预定SQL语句等),对于应用程序来说,它是只读的。外置缓存中存放的是数据库数据的副本,其作用和一级缓存类似.二级缓存除

了以内存作为存储介质外,还可以选用硬盘等外部存储设备。二级缓存称为进程级缓存或SessionFactory级缓存，它可以被所有session共享，它的生命周期

伴随着SessionFactory的生命周期存在和消亡。

MyBatis缓存

MyBatis 包含一个非常强大的查询缓存特性,它可以非常方便地配置和定制。MyBatis 3 中的缓存实现的很多改进都已经实现了,使得它更加强大而且易于配

置。

Hibernate的调优方案

制定合理的缓存策略；

尽量使用延迟加载特性；

采用合理的Session管理机制；

使用批量抓取，设定合理的批处理参数（batch\_size）;

进行合理的O/R映射设计

MyBatis同样具有二级缓存机制。 MyBatis可以进行详细的SQL优化设计

两者相同点

Hibernate与MyBatis都可以是通过SessionFactoryBuider由XML配置文件生成SessionFactory，然后由SessionFactory 生成Session，最后由Session来开启

执行事务和SQL语句。其中SessionFactoryBuider，SessionFactory，Session的生命周期都是差不多的。

Hibernate和MyBatis都支持JDBC和JTA事务处理。

Mybatis优势

MyBatis可以进行更为细致的SQL优化，可以减少查询字段。

MyBatis容易掌握，而Hibernate门槛较高。

Hibernate优势

Hibernate的DAO层开发比MyBatis简单，Mybatis需要维护SQL和结果映射。

Hibernate对对象的维护和缓存要比MyBatis好，对增删改查的对象的维护要方便。

Hibernate数据库移植性很好，MyBatis的数据库移植性不好，不同的数据库需要写不同SQL。

Hibernate有更好的二级缓存机制，可以使用第三方缓存。MyBatis本身提供的缓存机制不佳。

他人总结

Hibernate功能强大，数据库无关性好，O/R映射能力强，如果你对Hibernate相当精通，而且对Hibernate进行了适当的封装，那么你的项目整个持久层代码

会相当简单，需要写的代码很少，开发速度很快，非常爽。

Hibernate的缺点就是学习门槛不低，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡取得平衡，以及怎样用好Hibernate方面需要

你的经验和能力都很强才行。

iBATIS入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完

美。

iBATIS的缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不

太容易适应快速数据库修改

一、由于对持久层封装过于完整，导致开发人员无法对SQL进行优化，无法灵活使用JDBC的原生SQL，Hibernate封装了JDBC，所以没有JDBC直接访问数据库效率高。要使用数据库的特定优化机制的时候，不适合用Hibernate

二、框架中使用ORM原则，导致配置过于复杂，一旦遇到大型项目，比如300张表以上，配置文件和内容是非常庞大的，另外，DTO满天飞，性能和维护问题随之而来

三、如果项目中各个表中关系复杂，表之间的关系很多，在很多地方把lazy都设置false，会导致数据查询和加载很慢，尤其是级联查询的时候。

四、Hibernate在批量数据处理时有弱势，对于批量的修改，删除，不适合用Hibernate,这也是ORM框架的弱点

# RequireJS、SeaJS 区别

（1）两者定位有差异。RequireJS 想成为浏览器端的模块加载器，同时也想成为 Rhino / Node 等环境的模块加载器。SeaJS 则专注于 Web 浏览器端，同时通过 Node 扩展的方式可以很方便跑在 Node 服务器端

（2）两者遵循的标准有差异。RequireJS 遵循的是 AMD（异步模块定义）规范，SeaJS 遵循的是 CMD （通用模块定义）规范。规范的不同，导致了两者 API 的不同。SeaJS 更简洁优雅。

（3）两者社区理念有差异。RequireJS 在尝试让第三方类库修改自身来支持 RequireJS，目前只有少数社区采纳。SeaJS 不强推，而采用自主封装的方式来“海纳百川”，目前已有较成熟的封装策略。

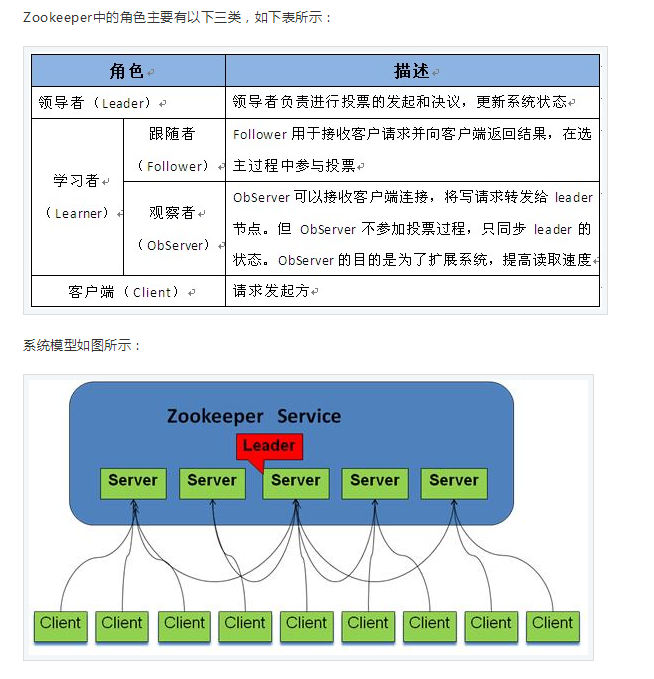
（4）两者代码质量有差异。RequireJS 是没有明显的 bug，SeaJS 是明显没有 bug。

（5）两者对调试等的支持有差异。SeaJS 通过插件，可以实现 Fiddler 中自动映射的功能，还可以实现自动 combo 等功能，非常方便便捷。RequireJS 无这方面的支持。

（6）两者的插件机制有差异。RequireJS 采取的是在源码中预留接口的形式，源码中留有为插件而写的代码。SeaJS 采取的插件机制则与 Node 的方式一致：开放自身.

# Zookeeper

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，它包含一个简单的原语集，分布式应用程序可以基于它实现同步服务，配置维护和命名服务等。Zookeeper是hadoop的一个子项目，其发展历程无需赘述。在分布式应用中，由于工程师不能很好地使用锁机制，以及基于消息的协调机制不适合在某些应用中使用，因此需要有一种可靠的、可扩展的、分布式的、可配置的协调机制来统一系统的状态。Zookeeper的目的就在于此。本文简单分析zookeeper的工作原理，对于如何使用zookeeper不是本文讨论的重点。



## 设计目的

1.最终一致性：client不论连接到哪个Server，展示给它都是同一个视图，这是zookeeper最重要的性能。

2 .可靠性：具有简单、健壮、良好的性能，如果消息m被到一台服务器接受，那么它将被所有的服务器接受。

3 .实时性：Zookeeper保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。但由于网络延时等原因，Zookeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用sync()接口。

4 .等待无关（wait-free）：慢的或者失效的client不得干预快速的client的请求，使得每个client都能有效的等待。

5.原子性：更新只能成功或者失败，没有中间状态。

6 .顺序性：包括全局有序和偏序两种：全局有序是指如果在一台服务器上消息a在消息b前发布，则在所有Server上消息a都将在消息b前被发布；偏序是指如果一个消息b在消息a后被同一个发送者发布，a必将排在b前面。

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的统治时期。低32位用于递增计数。

每个Server在工作过程中有三种状态：

LOOKING：当前Server不知道leader是谁，正在搜寻

LEADING：当前Server即为选举出来的leader

FOLLOWING：leader已经选举出来，当前Server与之同步

**集群管理**应用集群中，我们常常需要让每一个机器知道集群中（或依赖的其他某一个集群）哪些机器是活着的，并且在集群机器因为宕机，网络断链等原因能够不在人工介入的情况下迅速通知到每一个机器。  
  
Zookeeper同样很容易实现这个功能，比如我在zookeeper服务器端有一个znode叫/APP1SERVERS,那么集群中每一个机器启动的时候都去这个节点下创建一个EPHEMERAL类型的节点，比如server1创建/APP1SERVERS/SERVER1(可以使用ip,保证不重复)，server2创建/APP1SERVERS/SERVER2，然后SERVER1和SERVER2都watch /APP1SERVERS这个父节点，那么也就是这个父节点下数据或者子节点变化都会通知对该节点进行watch的客户端。因为EPHEMERAL类型节点有一个很重要的特性，就是客户端和服务器端连接断掉或者session过期就会使节点消失，那么在某一个机器挂掉或者断链的时候，其对应的节点就会消失，然后集群中所有对/APP1SERVERS进行watch的客户端都会收到通知，然后取得最新列表即可。  
  
另外有一个应用场景就是集群选master,一旦master挂掉能够马上能从slave中选出一个master,实现步骤和前者一样，只是机器在启动的时候在APP1SERVERS创建的节点类型变为EPHEMERAL\_SEQUENTIAL类型，这样每个节点会自动被编号

**解决什么问题？**

1.master挂机，传统做法备份必然是以前数据，该如何保证挂机数据与备份数据一致？

2.分布式系统如何实现对同一资源的访问，保证数据的强一致性？

3.集群中的worker挂了，传统做法是什么？zookeeper又是如何做的？

分布式系统的运行是很复杂的，因为涉及到了网络通信还有节点失效等不可控的情况。下面介绍在最传统的master-workers模型，主要可以会遇到什么问题，传统方法是怎么解决以及怎么用zookeeper解决。

Master节点管理

集群当中最重要的是Master，所以一般都会设置一台Master的Backup。

Backup会定期向Master获取Meta信息并且检测Master的存活性，一旦Master挂了，Backup立马启动，接替Master的工作自己成为Master，分布式的情况多种多样，因为涉及到了网络通信的抖动，针对下面的情况:

Backup检测Master存活性传统的就是定期发包，一旦一定时间段内没有收到响应就判定Master Down了，于是Backup就启动，如果Master其实是没有down，Backup收不到响应或者收到响应延迟的原因是因为网络阻塞的问题呢？Backup也启动了，这时候集群里就有了两个Master，很有可能部分workers汇报给Master，另一部分workers汇报给后来启动的Backup，这下子服务就全乱了。

Backup是定期同步Master中的meta信息，所以总是滞后的，一旦Master挂了，Backup的信息必然是老的，很有可能会影响集群运行状态。

解决问题:

Master节点高可用，并且保证唯一。

Meta信息的及时同步

zookeeper Master选举

zookeeper会分配给注册到它上面的客户端一个编号，并且zk自己会保证这个编号的唯一性和递增性，N多机器中只需选出编号最小的Client作为Master就行，并且保证这些机器的都维护一个一样的meta信息视图，一旦Master挂了，那么这N机器中编号最小的胜任Master，Meta信息是一致的。

配置文件管理

集群中配置文件的更新和同步是很频繁的，传统的配置文件分发都是需要把配置文件数据分发到每台worker上，然后进行worker的reload，这种方式是最笨的方式，结构很难维护，因为如果集群当中有可能很多种应用的配置文件要同步，而且效率很低，集群规模一大负载很高。还有一种就是每次更新把配置文件单独保存到一个数据库里面，然后worker端定期pull数据，这种方式就是数据及时性得不到同步。

解决问题:

统一配置文件分发并且及时让worker生效

zookeeper发布与订阅模型

发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到ZK节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，服务式服务框架的服务地址列表等就非常适合使用。

分布式锁

在一台机器上要多个进程或者多个线程操作同一资源比较简单，因为可以有大量的状态信息或者日志信息提供保证，比如两个A和B进程同时写一个文件，加锁就可以实现。但是分布式系统怎么办？需要一个三方的分配锁的机制，几百台worker都对同一个网络中的文件写操作，怎么协同？还有怎么保证高效的运行？

解决问题:

高效分布式的分布式锁

zookeeper分布式锁

分布式锁主要得益于ZooKeeper为我们保证了数据的强一致性，zookeeper的znode节点创建的唯一性和递增性能保证所有来抢锁的worker的原子性。

集群worker管理

集群中的worker挂了是很可能的，一旦workerA挂了，如果存在其余的workers互相之间需要通信，那么workers必须尽快更新自己的hosts列表，把挂了的worker剔除，从而不在和它通信，而Master要做的是把挂了worker上的作业调度到其他的worker上。同样的，这台worker重新恢复正常了，要通知其他的workers更新hosts列表。传统的作法都是有专门的监控系统，通过不断去发心跳包(比如ping)来发现worker是否alive，缺陷就是及时性问题，不能应用于在线率要求较高的场景

解决问题:

集群worker监控

zookeeper监控集群

利用zookeeper建立znode的强一致性，可以用于那种对集群中机器状态，机器在线率有较高要求的场景，能够快速对集群中机器变化作出响应。

# Metaq/Rocketmq/Kafka/ActiveMq/RabbitMQ

消息队列（Message Queue，简称MQ），从字面意思上看，本质是个队列，FIFO先入先出，只不过队列中存放的内容是message而已。其主要用途：不同进程Process/线程Thread之间通信。为什么会产生消息队列？这个问题问的好，我大概查了一下，没有查到最初产生消息队列的背景，但我猜测可能几个原因：

不同进程（process）之间传递消息时，两个进程之间耦合程度过高，改动一个进程，引发必须修改另一个进程，为了隔离这两个进程，在两进程间抽离出一层（一个模块），所有两进程之间传递的消息，都必须通过消息队列来传递，单独修改某一个进程，不会影响另一个；

不同进程（process）之间传递消息时，为了实现标准化，将消息的格式规范化了，并且，某一个进程接受的消息太多，一下子无法处理完，并且也有先后顺序，必须对收到的消息进行排队，因此诞生了事实上的消息队列；

　　不管到底是什么原因催生了消息队列，总之，上面两个猜测是其实际应用的典型场景。

　　为什么要用

通过消息队列，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务，改善网站系统的性能。在京东之类的电子商务网站促销活动中，合理地使用消息 队列，可以有效地抵御促销活动刚开始就开始大量涌入的订单对系统造成的冲击。

　　切合前一部分猜测的消息队列产生背景，其主要解决两个问题：

系统解耦：项目开始时，无法确定最终需求，不同进程间，添加一层，实现解耦，方便今后的扩展。

消息缓存：系统中，不同进程处理消息速度不同，MQ，可以实现不同Process之间的缓冲，即，写入MQ的速度可以尽可能地快，而处理消息的速度可以适当调整（或快、或慢）。

　　下面针对系统解耦、消息缓存两点，来分析实际应用消息队列过程中，可能遇到的问题。虚拟场景：Process\_A通过消息队列MQ\_1向Process\_B传递消息，几个问题：

针对MQ\_1中一条消息message\_1，如何确保Process\_B从MQ\_1中只取一次message\_1，不会重复多次取出message\_1？

如果MQ\_1中message\_1已经被Process\_B取出，正在处理的关键时刻，Process\_B崩溃了，哭啊，我的问题是，如果重启Process\_B，是否会丢失message\_1？

　　不要着急，阅读了下面的简要介绍后，水到渠成，上面几个问题就可以解决了。 消息队列有如下几个好处，这大都是由其系统解耦和消息缓存两点扩展而来的：

提升系统可靠性：

冗余：Process\_B崩溃之后，数据并不会丢失，因为MQ多采用put-get-delete模式，即，仅当确认message被完成处理之后，才从MQ中移除message；

可恢复：MQ实现解耦，部分进程崩溃，不会拖累整个系统瘫痪，例，Process\_B崩溃之后，Process\_A仍可向MQ中添加message，并等待Process\_B恢复；

可伸缩：有较强的峰值处理能力，通常应用会有突发的访问流量上升情况，使用足够的硬件资源时刻待命，空闲时刻较长，资源浪费，而消息队列却能够平滑峰值流量，缓解系统组件的峰值压力；

提升系统可扩展性：

调整模块：由于实现解耦，可以很容易调整，消息入队速率、消息处理速率、增加新的Process；

其他：

单次送达：保证MQ中一个message被处理一次，并且只被处理一次。本质：get获取一个message后，这一message即被预定，同一进程不会再次获取这一message；当且仅当进程处理完这一message后，MQ中会delete这个message。否则，过一段时间后，这一message自动解除被预订状态，进程能够重新预定这个message；

排序保证：即，满足队列的FIFO，先入先出策略；

异步通信：很多场景下，不会立即处理消息，这是，可以在MQ中存储message，并在某一时刻再进行处理；

数据流的阶段性能定位：获取用户某一操作的各个阶段（通过message来标识），捕获不同阶段的耗时，可用于定位系统瓶颈。

MetaQ（全称Metamorphosis）是一个高性能、高可用、可扩展的分布式消息中间件，思路起源于LinkedIn的Kafka，但并不是Kafka的一个Copy。MetaQ具有消息存储顺序写、吞吐量大和支持本地和XA事务等特性，适用于大吞吐量、顺序消息、广播和日志数据传输等场景，目前在淘宝和支付宝有着广泛的应用。

    Kafka是scala写的，我对scala不熟悉，并且在当时kafka整个社区的发展太缓慢了。

    有一些功能是kakfa没有实现，但是我们却需要，比如事务、多种offset存储、高可用方案(HA)等

ActiveMQ和HornetQ都是符合[Java](http://lib.csdn.net/base/java)EE中JMS规范的MQ实现，两者都是很优秀的开源MQ。同时两者也不局限在JMS规范，同时也支持其他一些MQ协议，如stomp协议、AMQP协议等。相比来说，就我当时了解的情况来看，HornetQ的性能会比ActiveMQ更强，因为HornetQ使用JNI基于异步IO做了更多优化，而对于MQ来说，最终的瓶颈都是落在IO存储上。MetaQ的性能是远远超过这两个MQ的，有一个网友做的比较可以说明一定问题（<http://www.blogjava.net/livery/articles/391595.html>），但是这不能说MetaQ比它们就更优秀，因为这跟它们的实现，和面对的场景有很大关系。

ActiveMQ和HornetQ，这两个MQ从诞生起就是为了企业应用而设计的，JMS规范本身也是企业应用系统的规范。这一套东西，我个人认为并不适合互联网应用。互联网应用通常面对的是海量的数据，并且通常对事务一致性的要求相对较弱，而企业应用对事务一致性的要求就相对很高。互联网应用为了处理大量请求，通常采用集群处理的方式，而JMS规范并不重视分布式应用。我说的这个集群不仅仅是服务端broker的集群，还包括生产者和消费者都可能是一个又一个集群，而传统的JMS规范是没有明确处理这些情况的。互联网应用还有一个问题是异构系统特别多，而JMS规范只是Java EE这个平台上的规范，对异构系统的接入也是一个比较麻烦的地方，不同的实现有很大的差异。  
  
综合来讲，HornetQ和ActiveMQ是为了企业级应用设计的消息中间件，而MetaQ从一开始就是为了大规模互联网应用设计的消息中间件，两者面对的场景和需求不同。开发者可根据实际的需求，选择合适的产品。

Kafka的吞吐量高达17.3w/s，不愧是高吞吐量消息中间件的行业老大。这主要取决于它的队列模式保证了写磁盘的过程是线性IO。此时broker磁盘IO已达瓶颈。

RocketMQ也表现不俗，吞吐量在11.6w/s，磁盘IO %util已接近100%。RocketMQ的消息写入内存后即返回ack，由单独的线程专门做刷盘的操作，所有的消息均是顺序写文件。

RabbitMQ的吞吐量5.95w/s，CPU资源消耗较高。它支持AMQP协议，实现非常重量级，为了保证消息的可靠性在吞吐量上做了取舍。我们还做了RabbitMQ在消息持久化场景下的性能测试，吞吐量在2.6w/s左右。

Kafka是LinkedIn开源的分布式发布-订阅消息系统，目前归属于Apache定级项目。Kafka主要特点是基于Pull的模式来处理消息消费，追求高吞吐量，一开始的目的就是用于日志收集和传输。0.8版本开始支持复制，不支持事务，对消息的重复、丢失、错误没有严格要求，适合产生大量数据的互联网服务的数据收集业务。

RabbitMQ是使用Erlang语言开发的开源消息队列系统，基于AMQP协议来实现。AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。AMQP协议更多用在企业系统内，对数据一致性、稳定性和可靠性要求很高的场景，对性能和吞吐量的要求还在其次。

RocketMQ是阿里开源的消息中间件，它是纯Java开发，具有高吞吐量、高可用性、适合大规模分布式系统应用的特点。RocketMQ思路起源于Kafka，但并不是Kafka的一个Copy，它对消息的可靠传输及事务性做了优化，目前在阿里集团被广泛应用于交易、充值、流计算、消息推送、日志流式处理、binglog分发等场景。

kafka的优点：

1.主要是用来解决百万级别的数据中生产者和消费者之间数据传输的问题

2.可以将一条数据提供给多个接收这做不同的处理

3.当两个系统是隔绝的，无法通信的时候，如果想要他们通信就需要重新构建其中的一个工程，而kafka实现了生产者和消费者之间的无缝对接。

4.[大数据](http://lib.csdn.net/base/hadoop)时代，最重要的是数据的收集和分析，这些数据包括：

1）.用户的行为数据

2）.应用工程的性能数据

3）.日志的用户活动数据等

5.kafka提供了系统之间的消息通信，对于生产者而言，只关注与把消息发送的kafka上，而并不关心这个消息是被谁消费的（kafka相当于消息的代理者）。

6.kafka是一个开源的消息发布和订阅系统，主要用于以下场景中：

1）.持续的消息：为了从大数据中派生出有用的数据，任何数据的丢失都会影响生成的结果，kafka提供了一个复杂度为O(1)的磁盘结构存储数据，即使是对于TB级别的数据都是提供了一个常量时间性能。

2）.高吞吐量：keep big data in mind，kafka采用普通的硬件支持每秒百万级别的吞吐量

3）.分布式：明确支持消息的分区，通过kafka服务器和消费者机器的集群分布式消费，维持每一个分区是有序的。

4）.支持多种语言：[Java](http://lib.csdn.net/base/java)、.net、[PHP](http://lib.csdn.net/base/php)、ruby、[Python](http://lib.csdn.net/base/python)。

5）.实时性：消息被生成者线程生产就能马上被消费者线程消费，这种特性和事件驱动的系统是相似的。

# Dubbo

Dubbo是Alibaba开源的分布式服务框架，我们可以非常容易地通过Dubbo来构建分布式服务，并根据自己实际业务应用场景来选择合适的集群容错模式，这个对于很多应用都是迫切希望的，只需要通过简单的配置就能够实现分布式服务调用，也就是说服务提供方（Provider）发布的服务可以天然就是集群服务，比如，在实时性要求很高的应用场景下，可能希望来自消费方（Consumer）的调用响应时间最短，只需要选择Dubbo的Forking Cluster模式配置，就可以对一个调用请求并行发送到多台对等的提供方（Provider）服务所在的节点上，只选择最快一个返回响应的，然后将调用结果返回给服务消费方（Consumer），显然这种方式是以冗余服务为基础的，需要消耗更多的资源，但是能够满足高实时应用的需求。  
有关Dubbo服务框架的简单使用，可以参考我的其他两篇文章（《基于Dubbo的Hessian协议实现远程调用》，《Dubbo实现RPC调用使用入门》，后面参考链接中已给出链接），这里主要围绕Dubbo分布式服务相关配置的使用来说明与实践。

## 1. Dubbo是什么？

Dubbo是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案。简单的说，dubbo就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有dubbo这样的分布式服务框架的需求，并且本质上是个服务调用的东东，**说白了就是个远程服务调用的分布式框架（告别Web Service模式中的WSdl，以服务者与消费者的方式在dubbo上注册）**  
其核心部分包含:  
1. 远程通讯: 提供对多种基于长连接的NIO框架抽象封装，包括多种线程模型，序列化，以及“请求-响应”模式的信息交换方式。  
2. 集群容错: 提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。  
3. 自动发现: 基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

## 2. Dubbo能做什么？

1.透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。        
2.软负载均衡及容错机制，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。  
3. 服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。  
  
Dubbo采用全[**spring**](http://lib.csdn.net/base/javaee)配置方式，透明化接入应用，对应用没有任何API侵入，只需用Spring加载Dubbo的配置即可，Dubbo基于Spring的Schema扩展进行加载。

**之前使用Web Service，我想**[**测试**](http://lib.csdn.net/base/softwaretest)**接口可以通过模拟消息的方式通过soapui或LR进行功能测试或性能测试。但现在使用Dubbo，接口之间不能直接交互，我尝试通过模拟消费者地址测试，结果不堪入目，再而使用jmeter通过junit进行测试，但还是需要往dubbo上去注册，如果再不给提供源代码的前提下，这个测试用例不好写啊....**

# zookeeper和dubbo

1.   Zookeeper的作用：

        zookeeper用来注册服务和进行负载均衡，哪一个服务由哪一个机器来提供必需让调用者知道，简单来说就是ip地址和服务名称的对应关系。当然也可以通过硬编码的方式把这种对应关系在调用方业务代码中实现，但是如果提供服务的机器挂掉调用者无法知晓，如果不更改代码会继续请求挂掉的机器提供服务。zookeeper通过心跳机制可以检测挂掉的机器并将挂掉机器的ip和服务对应关系从列表中删除。至于支持高并发，简单来说就是横向扩展，在不更改代码的情况通过添加机器来提高运算能力。通过添加新的机器向zookeeper注册服务，服务的提供者多了能服务的客户就多了。

2.  dubbo：

      是管理中间层的工具，在业务层到数据仓库间有非常多服务的接入和服务提供者需要调度，dubbo提供一个框架解决这个问题。

      注意这里的dubbo只是一个框架，至于你架子上放什么是完全取决于你的，就像一个汽车骨架，你需要配你的轮子引擎。这个框架中要完成调度必须要有一个分布式的注册中心，储存所有服务的元数据，你可以用zk，也可以用别的，只是大家都用zk。  
  
3. zookeeper和dubbo的关系：  
      Dubbo的将注册中心进行抽象，是得它可以外接不同的存储媒介给注册中心提供服务，有ZooKeeper，Memcached，Redis等。  
      引入了ZooKeeper作为存储媒介，也就把ZooKeeper的特性引进来。首先是负载均衡，单注册中心的承载能力是有限的，在流量达到一定程度的时候就需要分流，负载均衡就是为了分流而存在的，一个ZooKeeper群配合相应的Web应用就可以很容易达到负载均衡；资源同步，单单有负载均衡还不够，节点之间的数据和资源需要同步，ZooKeeper集群就天然具备有这样的功能；命名服务，将树状结构用于维护全局的服务地址列表，服务提供者在启动的时候，向ZK上的指定节点/dubbo/${serviceName}/providers目录下写入自己的URL地址，这个操作就完成了服务的发布。其他特性还有Mast选举，分布式锁等。

# Jetty和tomcat的比较

相同点：

1.      Tomcat和Jetty都是一种Servlet引擎，他们都支持标准的servlet规范和JavaEE的规范。

不同点：

1.      [**架构**](http://lib.csdn.net/base/architecture)比较

Jetty的架构比Tomcat的更为简单

Jetty的架构是基于Handler来实现的，主要的扩展功能都可以用Handler来实现，扩展简单。

Tomcat的架构是基于容器设计的，进行扩展是需要了解Tomcat的整体设计结构，不易扩展。

2.      性能比较

Jetty和Tomcat性能方面差异不大

Jetty可以同时处理大量连接而且可以长时间保持连接，适合于web聊天应用等等。

Jetty的架构简单，因此作为服务器，Jetty可以按需加载组件，减少不需要的组件，减少了服务器内存开销，从而提高服务器性能。

Jetty默认采用NIO结束在处理I/O请求上更占优势，在处理静态资源时，性能较高

Tomcat适合处理少数非常繁忙的链接，也就是说链接生命周期短的话，Tomcat的总体性能更高。

Tomcat默认采用BIO处理I/O请求，在处理静态资源时，性能较差。

3.      其它比较

Jetty的应用更加快速，修改简单，对新的Servlet规范的支持较好。

Tomcat目前应用比较广泛，对JavaEE和Servlet的支持更加全面，很多特性会直接集成进来。

# 游戏SDK

sdk是 Software Development Kit （[软件开发工具包](https://www.baidu.com/s?wd=%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91%E5%B7%A5%E5%85%B7%E5%8C%85&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkmWIWPAu9rjfkPW6znynY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjRkPHbzn1n3)）的缩写。

一般都是一些软件工程师为特定的软件包、软件框架、硬件平台、[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkmWIWPAu9rjfkPW6znynY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjRkPHbzn1n3)等建立应用软件时的开发工具的集合。

[软件开发工具包](https://www.baidu.com/s?wd=%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91%E5%B7%A5%E5%85%B7%E5%8C%85&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkmWIWPAu9rjfkPW6znynY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjRkPHbzn1n3)广义上指辅助开发某一类软件的相关文档、范例和工具的集合。

比如在游戏开发中，就包含有很多的sdk。

再如，开发安卓应用时，也都需要下载安装Android SDK

# Spark

## 什么是Spark

[Apache Spark](https://spark.apache.org/)是一个围绕速度、易用性和复杂分析构建的大数据处理框架。最初在2009年由加州大学伯克利分校的AMPLab开发，并于2010年成为Apache的开源项目之一。

与Hadoop和Storm等其他大数据和MapReduce技术相比，Spark有如下优势。

首先，Spark为我们提供了一个全面、统一的框架用于管理各种有着不同性质（文本数据、图表数据等）的数据集和数据源（批量数据或实时的流数据）的大数据处理的需求。

Spark可以将Hadoop集群中的应用在内存中的运行速度提升100倍，甚至能够将应用在磁盘上的运行速度提升10倍。

Spark让开发者可以快速的用Java、Scala或Python编写程序。它本身自带了一个超过80个高阶操作符集合。而且还可以用它在shell中以交互式地查询数据。

除了Map和Reduce操作之外，它还支持SQL查询，流数据，机器学习和图表数据处理。开发者可以在一个数据管道用例中单独使用某一能力或者将这些能力结合在一起使用。

在这个Apache Spark文章系列的第一部分中，我们将了解到什么是Spark，它与典型的MapReduce解决方案的比较以及它如何为大数据处理提供了一套完整的工具。

## Hadoop和Spark

Hadoop这项大数据处理技术大概已有十年历史，而且被看做是首选的大数据集合处理的解决方案。MapReduce是一路计算的优秀解决方案，不过对于需要多路计算和算法的用例来说，并非十分高效。数据处理流程中的每一步都需要一个Map阶段和一个Reduce阶段，而且如果要利用这一解决方案，需要将所有用例都转换成MapReduce模式。

在下一步开始之前，上一步的作业输出数据必须要存储到分布式文件系统中。因此，复制和磁盘存储会导致这种方式速度变慢。另外Hadoop解决方案中通常会包含难以安装和管理的集群。而且为了处理不同的大数据用例，还需要集成多种不同的工具（如用于机器学习的Mahout和流数据处理的Storm）。

如果想要完成比较复杂的工作，就必须将一系列的MapReduce作业串联起来然后顺序执行这些作业。每一个作业都是高时延的，而且只有在前一个作业完成之后下一个作业才能开始启动。

而Spark则允许程序开发者使用有向无环图（[DAG](http://en.wikipedia.org/wiki/Directed_acyclic_graph)）开发复杂的多步数据管道。而且还支持跨有向无环图的内存数据共享，以便不同的作业可以共同处理同一个数据。

Spark运行在现有的Hadoop分布式文件系统基础之上（[HDFS](http://wiki.apache.org/hadoop/HDFS)）提供额外的增强功能。它支持[将Spark应用部署到](http://databricks.com/blog/2014/01/21/Spark-and-Hadoop.html)现存的Hadoop v1集群（with SIMR – Spark-Inside-MapReduce）或Hadoop v2 YARN集群甚至是[Apache Mesos](http://mesos.apache.org/)之中。

我们应该将Spark看作是Hadoop MapReduce的一个替代品而不是Hadoop的替代品。其意图并非是替代Hadoop，而是为了提供一个管理不同的大数据用例和需求的全面且统一的解决方案。

## Spark特性

Spark通过在数据处理过程中成本更低的洗牌（Shuffle）方式，将MapReduce提升到一个更高的层次。利用内存数据存储和接近实时的处理能力，Spark比其他的大数据处理技术的性能要快很多倍。

Spark还支持大数据查询的延迟计算，这可以帮助优化大数据处理流程中的处理步骤。Spark还提供高级的API以提升开发者的生产力，除此之外还为大数据解决方案提供一致的体系架构模型。

Spark将中间结果保存在内存中而不是将其写入磁盘，当需要多次处理同一数据集时，这一点特别实用。Spark的设计初衷就是既可以在内存中又可以在磁盘上工作的执行引擎。当内存中的数据不适用时，Spark操作符就会执行外部操作。Spark可以用于处理大于集群内存容量总和的数据集。

Spark会尝试在内存中存储尽可能多的数据然后将其写入磁盘。它可以将某个数据集的一部分存入内存而剩余部分存入磁盘。开发者需要根据数据和用例评估对内存的需求。Spark的性能优势得益于这种内存中的数据存储。

Spark的其他特性包括：

* 支持比Map和Reduce更多的函数。
* 优化任意操作算子图（operator graphs）。
* 可以帮助优化整体数据处理流程的大数据查询的延迟计算。
* 提供简明、一致的Scala，Java和Python API。
* 提供交互式Scala和Python Shell。目前暂不支持Java。

# sql优化器

Oracle有两种优化器：RBO和CBO。 RBO的最大的问题在于它是靠硬编码在ORACLE数据库代码中的一系列规定的规则来决定目标SQL的执行计划的，而并没有考虑目标SQL中所涉及的对象的时间数据量，实际数据分布情况，这样一旦规定规则并不适用于该SQL中所涉及的实际对象时，RBO根据规定规则产生的执行计划就很可能不是当前情况下的最优执行计划了。  
 下面我们来看如下的例子：  
 select \* from EMP\_TEMP where manager\_id=100;  
假设在EMP\_TEMP的manager\_id上事先有名为IDX\_MGR\_TEMP的单键值B数索引，如果我们用的是RBO，则不管EMP\_TEMP的数据量多大，也不管MANAGER\_ID的数据分布如何，ORACLE执行的时候始终会选择做对IDX\_MGR\_TEMP的范围索引扫描，并回表取得EMP\_TEMP中的记录。ORACLE是不会选择[全表扫描](https://www.baidu.com/s?wd=%E5%85%A8%E8%A1%A8%E6%89%AB%E6%8F%8F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)EMP\_TEMP表的，因为对于RBO而言，[全表扫描](https://www.baidu.com/s?wd=%E5%85%A8%E8%A1%A8%E6%89%AB%E6%8F%8F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)的等级值要高于索引范围扫描值的等级值。  
RBO的这种选择在表EMP\_TEMP的数据量不大，而且满足manager\_id=10的条件的记录少的情况下是影响不大的，如果表EMP\_TEMP的数据量非常大，例如1000万条记录，  
而且这1000万条记录的MANAGER\_ID的值都是100，在这种极端的情况下，如果是RBO,显然它[任然](https://www.baidu.com/s?wd=%E4%BB%BB%E7%84%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)用IDX\_MGR\_TEMP索引范围扫描，这个时候性能肯定是很差的。因为相当于以单块顺序扫描所有的1000万行索引，然后再回表1000万次。显然没有使用多块以[全表扫描](https://www.baidu.com/s?wd=%E5%85%A8%E8%A1%A8%E6%89%AB%E6%8F%8F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)方式直接扫描表EMP\_TEMP的执行效率高。所以为了解决RBO的这个先天的缺陷，从ORACLE 7开始，ORACLE就引入了CBO。CBO在选择目标SQL的执行计划时，是用执行成本作为判断原则的。CBO会从目标SQL诸多可能的执行路径中选择一条成本值最小的执行路径作为其执行计划，各条执行路径的成本是根据目标[SQL语句](https://www.baidu.com/s?wd=SQL%E8%AF%AD%E5%8F%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)所涉及的表，索引，列等相关对象的[统计信息](https://www.baidu.com/s?wd=%E7%BB%9F%E8%AE%A1%E4%BF%A1%E6%81%AF&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)计算出来的。这些信息存储在ORACLE的数据库的数据字典里，且从多个维度描述了ORACLE数据库里相关对象的实际数据量，实际数据分布等详细信息。  
NOTE:ORACLE在对一条执行路径计算成本时，并不一定从头到尾完整计算完，只是要ORACLE在计算过程中发现算出来的部分成本值已经大于之前保存下来的到目前为止的最小成本值，就会马上终止对当前执行路径成本值的计算，并转而开始计算下一条新的执行路径的成本。这个过程会一直持续下去，直到目标SQL的给各个可能的执行路径全部计算完毕或已经达到预先定义好的待计算的执行路径数量的阀值。   
RBO是根据硬编码在ORACLE数据库中来决定目标SQL的执行计划的，并没有考虑目标SQL所所涉及的对象的实际数据量，实际分布情况等。而CBO则恰恰相反，它会根据目标SQL的相关的对象的实际数据量，实际数据分布情况的[统计信息](https://www.baidu.com/s?wd=%E7%BB%9F%E8%AE%A1%E4%BF%A1%E6%81%AF&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)来决定其执行计划，即意味着CBO是随着目标SQL中所涉及的对象的[统计信息](https://www.baidu.com/s?wd=%E7%BB%9F%E8%AE%A1%E4%BF%A1%E6%81%AF&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)的变化而变化的。这就意味着只有统计信息相对准确，则用CBO来解析目标SQL会比同等条件下的RBO来解析得到正确执行计划的概率要高。  
当然CBO并不是完美的，它的缺陷主要表现在：  
1，CBO会默认目标[SQL语句](https://www.baidu.com/s?wd=SQL%E8%AF%AD%E5%8F%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-mhRkm1T4nW--nH9bnvn0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtvrjc4njTkPjR4rHTkn1R4n1c)的WHERE条件中出现的各个列之间是独立的，没有关系的。  
2，CBO会假设所有的目标SQL都是单独执行的，并且互不干扰。  
3，CBO对直方图统计信息有诸多限制。  
4，CBO在解析多个表关联的目标SQL时，可能会漏掉正确的执行计划。

# sql执行引擎



# 分布式计算

分布式计算简单来说，是把一个大计算任务拆分成多个小计算任务分布到若干台机器上去计算，然后再进行结果汇总。 目的在于分析计算海量的数据，从雷达监测的海量历史信号中分析异常信号(外星文明)，淘宝双十一实时计算各地区的消费习惯等。

海量计算最开始的方案是提高单机计算性能，如大型机，后来由于数据的爆发式增长、单机性能却跟不上，才有分布式计算这种妥协方案。 因为计算一旦拆分，问题会变得非常复杂，像一致性、数据完整、通信、容灾、任务调度等问题也都来了。

举个例子，产品要求从数据库中100G的用户购买数据，分析出各地域的消费习惯金额等。 如果没什么时间要求，程序员小明就写个对应的业务处理服务程序，部署到服务器上，让它慢慢跑就是了，小明预计10个小时能处理完。 后面产品嫌太慢，让小明想办法加快到3个小时。  
平常开发中类似的需求也很多，总结出来就是，数据量大、单机计算慢。 如果上Hadoop、storm之类成本较高、而且有点大才小用。 当然让老板买更好的服务器配置也是一种办法。

## 利用分片算法

小明作为一个有追求有理想的程序员，决定用介于单机计算和成熟计算框架的过度解决方案，这样成本和需求都能满足了。 分布式计算的核心在于计算任务拆分，如果数据能以水平拆分的方式，分布到5台机器上，每台机器只计算自身的1/5数据，这样即能在3小时内完成产品需求了。

如上所述，小明需要把这些数据按照一定维度进行划分。 按需求来看以用户ID划分最好，由于用户之间没有状态上的关联，所以也不需要事务性及二次迭代计算。 小明用简单的hash取模对id进行划分。

f(memberid) % 5 = ServerN

这样程序可以分别部署到5台机器上，然后程序按照配置只取对应余数的用户id，计算出结果并入库。 这种方式多机之间毫无关联，不需要进行通信，可以避免很多问题。 机器上的程序本身也不具备分布式的特性，它和单机一样，只计算自身获取到的数据即可，所以如果某台机器上程序崩溃的话，处理方式和单机一样，比如记录下处理进度，下次从当前进度继续进行后续计算。

## 利用消息队列

使用分片方式相对比较简单，但有如下不足之处。

* 它不具有负载均衡的能力，如果某台机器配置稍好点，它可能最先计算完，然后空闲等待着。也有可能是某些用户行为数据比较少，导致计算比较快完成。
* 还有一个弊端就是每台机器上需要手动更改对应的配置， 这样的话多台机器上的程序不是完全一样的，这样可以用远程配置动态修改的办法来解决。

小明这种方式引入了个第三方，消息队列。 小明先用一个单独的程序把用户信息推送到消息队列里去，然后各台机器分别取消费这个队列。 于是就有了3个角色：

* 推送消息的，简称Master。
* 消息队列，这里以Rabbitmq为例。
* 各个处理程序，简称Worker或Slave都行。

虽然仅仅引入了个第三方，但它已经具备了分布式计算的很多特性。

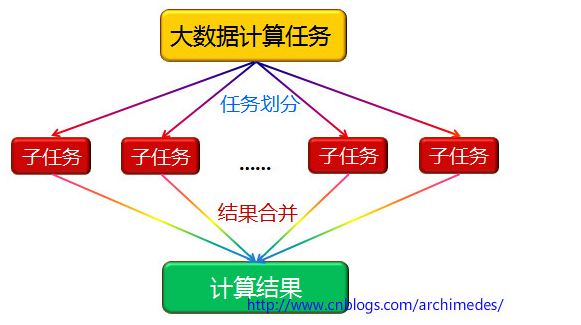
1. 计算任务分发。 Master把需要计算的用户数据，不断的推送消息队列。
2. 程序一致性。 Worker订阅相同的消息队列即可，无需更改程序代码。
3. 任意扩容。 由于程序完全一样，意味着如果想要加快速度，重复部署一份程序到新机器即可。 当然这是理论上的，实际当中会受限于消息队列、数据库存储等。
4. 容灾性。 如果5台中某一台程序挂了也不影响，利用Rabbitmq的消息确认机制，机器崩溃时正在计算的那一条数据会在超时，在其他节点上进行消费处理。

Hadoop介绍已经相当多了，这里简述下比如:"Hadoop是一套海量数据计算存储的基础平台架构"，分析下这句话。

* 其中计算指的是MapReduce，这是做分布式计算用的。
* 存储指的是HDFS，基于此上层的有HBase、Hive，用来做数据存储用的。
* 平台，指可以给多个用户使用，比如小明有一计算需求，他只需要按照对应的接口编写业务逻辑即可，然后把程序以包的形式发布到平台上，平台进行分配调度计算等。 而上面小明的分布式计算设计只能给自己使用，如果另外有小华要使用就需要重新写一份，然后单独部署，申请机器等。Hadoop最大的优势之一就在于提供了一套这样的完整解决方案。

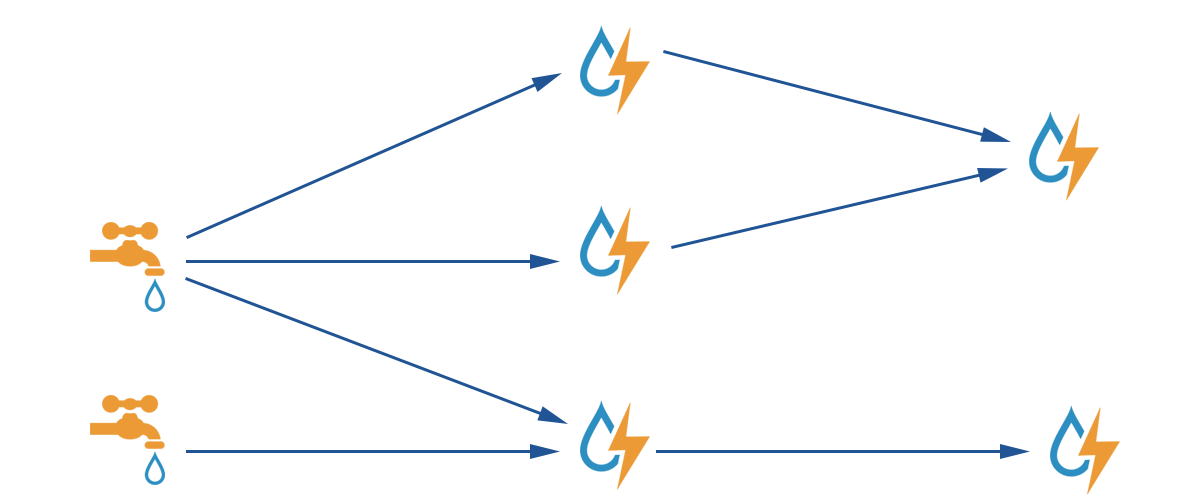
下面找了介绍Hadoop的概览图，跟小明的设计做对比下:

* 图中“大数据计算任务” 对应小明的100G用户数据的计算任务。
* ”任务划分“ 对应Master和消息队列。
* “子任务” 对应Worker的业务逻辑。
* ”结果合并“ 对应把每个worker的计算结果入库。
* “计算结果” 对应入库的用户消费习惯数据。

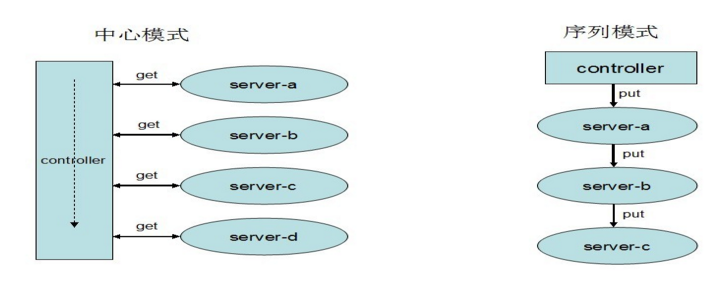


## 离线计算

通常称Mapreduce及小和尚这种计算为离线计算，因为它对已经持久化的文件数据进行计算，不能实时响应。 还有个原因就是它的处理速度比较慢，它的输入和输出源都是基于HDFS设计，如果数据不是一开始就写入到HDFS上，就会涉及到数据导入导出，这部分相对耗费时间。 而且它的数据流动是基于文件系统的，Map部分输出的数据不是直接传送到Reduce部分，而是先写入HDFS再进行传送。

处理速度慢也是Mapreduce的不足之处，促使了后面实时计算的诞生。  
另外个缺点是Mapreduce的计算任务流比较单一，它只有Map、Reduce两部分。 简单的可以只写一部分逻辑来解决，如果想拆分成多个部分，如逻辑A、逻辑B、逻辑C等， 而且一部分计算逻辑依赖上一次计算结果的话，MapReduce处理起来就比较困难了。 像storm框架解决此类问题的方案，也称为流式计算，下一章继续补充。   


# 流式计算



两种模式的优缺点

中心模式

get用阻塞方式实现（主线程收/回包，工作线程去阻塞get）

优点：实现简单

缺点：同一时刻并发数==线程数

get用非阻塞方式实现（只需一个线程）

优点：并发数高

缺点：容易出错，上一篇文章有详细说明（double-free,mem-leak，引用栈上空间）

本身的优点：

模块天生隔离，可重用性高

序列模式

缺点

上下游的耦合，可重用性不高

上下游处理速度不匹配导致cache数据

下游down机，上游处理麻烦

优点

只要还有任务在处理队列中，消费者都会及时从上游获得任务及时处理

两种模式的IO

中心模式IO次数看上去要比序列模式少，但这并不是它总体比序列模式快的原因；因为还有IO大小的问题，我们看下下面的列子

如果server-c需要使用server-a,server-b两个模块的处理结果，中心模式直接在内存里，木有网络IO；单序列模式需要两次IO才能把server-a的结果发给server-c;

看上去中心模式不存在：server-a,server-b速度处理速度不匹配导致的cache；但实际上它的controller如果不控制最大并发数，那么同样会把cache堆积到内存里；

# 流式大数据处理的三种框架：Storm，Spark和Samza

发表于2015-03-09 09:47| 次阅读| 来源javacodegeeks| 0 条评论| 作者Tony Sicilian

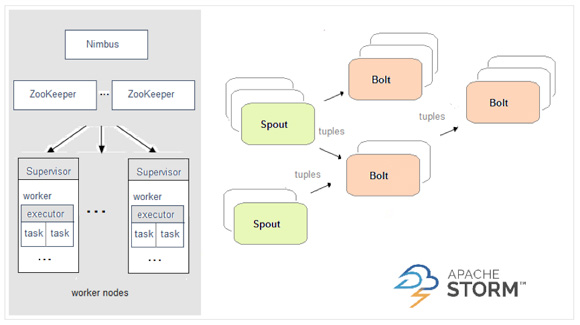
[Storm](http://www.csdn.net/tag/storm/news)[Samza](http://www.csdn.net/tag/samza/news)[Spark](http://www.csdn.net/tag/spark/news)[流计算](http://www.csdn.net/tag/%E6%B5%81%E8%AE%A1%E7%AE%97/news)[大数据](http://www.csdn.net/tag/%E5%A4%A7%E6%95%B0%E6%8D%AE/news)

**摘要：**许多分布**式计算**系统都可以实时或接近实时地处理大数据流。本文将对Storm、Spark和Samza等三种Apache**框架**分别进行简单介绍，然后尝试快速、高度概述其异同。

许多分布**式计算**系统都可以实时或接近实时地处理大数据流。本文将对三种Apache**框架**分别进行简单介绍，然后尝试快速、高度概述其异同。

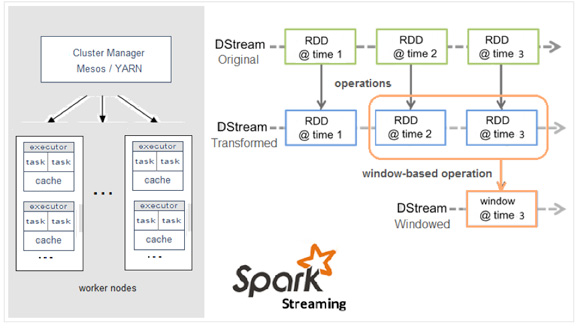
**Apache Storm**

在[Storm](https://storm.apache.org/)中，先要设计一个用于实时计算的图状结构，我们称之为拓扑（topology）。这个拓扑将会被提交给集群，由集群中的主控节点（master node）分发代码，将任务分配给工作节点（worker node）执行。一个拓扑中包括spout和bolt两种角色，其中spout发送消息，负责将数据流以tuple元组的形式发送出去；而bolt则负责转换这些数据流，在bolt中可以完成计算、过滤等操作，bolt自身也可以随机将数据发送给其他bolt。由spout发射出的tuple是不可变数组，对应着固定的键值对。



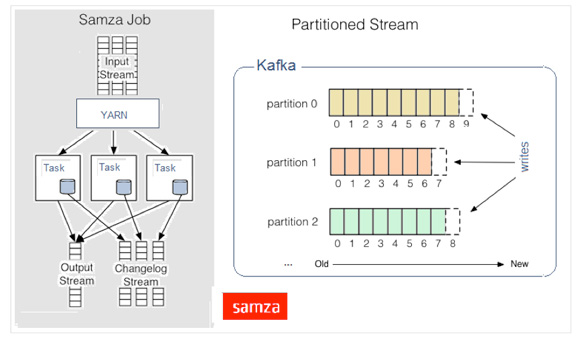
**Apache Spark**

[Spark Streaming](https://spark.apache.org/streaming/)是核心Spark API的一个扩展，它并不会像Storm那样一次一个地处理数据流，而是在处理前按时间间隔预先将其切分为一段一段的批处理作业。Spark针对持续性数据流的抽象称为DStream（DiscretizedStream），一个DStream是一个微批处理（micro-batching）的RDD（弹性分布式数据集）；而RDD则是一种分布式数据集，能够以两种方式并行运作，分别是任意函数和滑动窗口数据的转换。



**Apache Samza**

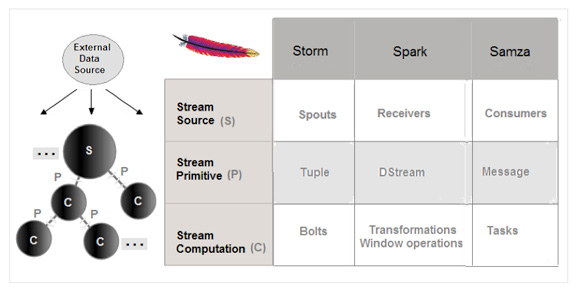
[Samza](http://samza.apache.org/)处理数据流时，会分别按次处理每条收到的消息。Samza的流单位既不是元组，也不是Dstream，而是一条条消息。在Samza中，数据流被切分开来，每个部分都由一组只读消息的有序数列构成，而这些消息每条都有一个特定的ID（offset）。该系统还支持批处理，即逐次处理同一个数据流分区的多条消息。Samza的执行与数据流模块都是可插拔式的，尽管Samza的特色是依赖Hadoop的Yarn（另一种资源调度器）和Apache Kafka。



**共同之处**

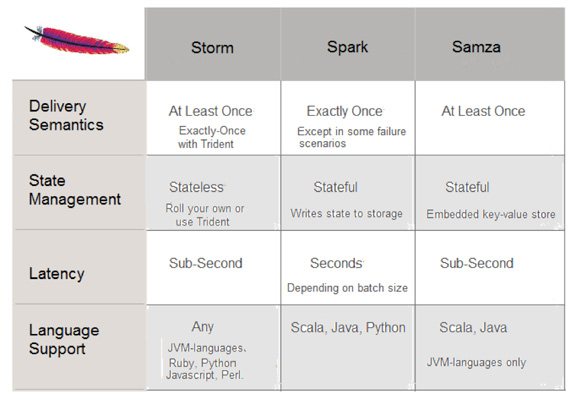
以上三种实时计算系统都是开源的分布式系统，具有低延迟、可扩展和容错性诸多优点，它们的共同特色在于：允许你在运行数据流代码时，将任务分配到一系列具有容错能力的计算机上并行运行。此外，它们都提供了简单的API来简化底层实现的复杂程度。

三种**框架**的术语名词不同，但是其代表的概念十分相似：



**对比图**

下面表格总结了一些不同之处：



数据传递形式分为三大类：

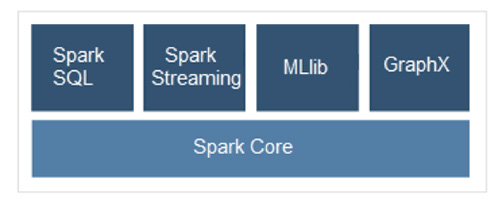
1. 最多一次（At-most-once）：消息可能会丢失，这通常是最不理想的结果。
2. 最少一次（At-least-once）：消息可能会再次发送（没有丢失的情况，但是会产生冗余）。在许多用例中已经足够。
3. 恰好一次（Exactly-once）：每条消息都被发送过一次且仅仅一次（没有丢失，没有冗余）。这是最佳情况，尽管很难保证在所有用例中都实现。

另一个方面是状态管理：对状态的存储有不同的策略，Spark Streaming将数据写入分布式文件系统中（例如HDFS）；Samza使用嵌入式键值存储；而在Storm中，或者将状态管理滚动至应用层面，或者使用更高层面的抽象Trident。

**用例**

这三种**框架**在处理连续性的大量实时数据时的表现均出色而高效，那么使用哪一种呢？选择时并没有什么硬性规定，最多就是几个指导方针。

如果你想要的是一个允许增量计算的高速事件处理系统，Storm会是最佳选择。它可以应对你在客户端等待结果的同时，进一步进行分布**式计算**的需求，使用开箱即用的分布式RPC（DRPC）就可以了。最后但同样重要的原因：Storm使用Apache Thrift，你可以用任何编程语言来编写拓扑结构。如果你需要状态持续，同时/或者达到恰好一次的传递效果，应当看看更高层面的Trdent API，它同时也提供了微批处理的方式。



**使用Storm的公司有：Twitter，雅虎，Spotify还有The Weather Channel等。**

说到微批处理，如果你必须有状态的计算，恰好一次的递送，并且不介意高延迟的话，那么可以考虑Spark Streaming，特别如果你还计划图形操作、机器学习或者访问SQL的话，Apache Spark的stack允许你将一些library与数据流相结合（Spark SQL，Mllib，GraphX），它们会提供便捷的一体化编程模型。尤其是数据流算法（例如：K均值流媒体）允许Spark实时决策的促进。

**使用Spark的公司有：亚马逊，雅虎，NASA JPL，eBay还有百度等。**

如果你有大量的状态需要处理，比如每个分区都有许多十亿位元组，那么可以选择Samza。由于Samza将存储与处理放在同一台机器上，在保持处理高效的同时，还不会额外载入内存。这种**框架**提供了灵活的可插拔API：它的默认execution、消息发送还有存储引擎操作都可以根据你的选择随时进行替换。此外，如果你有大量的数据流处理阶段，且分别来自不同代码库的不同团队，那么Samza的细颗粒工作特性会尤其适用，因为它们可以在影响最小化的前提下完成增加或移除的工作。

**使用Samza的公司有：LinkedIn，Intuit，Metamarkets，Quantiply，Fortscale等。**

**结论**

本文中我们只对这三种Apache**框架**进行了简单的了解，并未覆盖到这些**框架**中大量的功能与更多细微的差异。同时，文中这三种**框架**对比也是受到限制的，因为这些**框架**都在一直不断的发展，这一点是我们应当牢记的。

# 架构模式

1 横向切分：水平分表，在数据库层，有些表非常大，数据量在亿级，建立分区表,读写分离。主从数据库服务器。

2 纵向切分：垂直分库，将整个技术部按业务和平台拆分为部门，订单的，会员的，商家的。每个部门有自己的web服务器集群，数据库服务器集群

，通过同一个网站访问的链接可能来自于不同的服务器和数据库，对网站及底层对数据库的访问被分配到了不同的服务器集群,这个便是典型

的按业务做的垂直拆分，每个部门的服务器在hold不住时，会有弹性的扩展，这便是水平扩展。

3 分布式：分层和分割的主要目的是为了切分后的模块便于分布式部署，类似于微服务拆分。常用方案：

分布式应用和服务

分布式静态资源

分布式数据和存储

分布式计算

分布式配置，分布式锁，分布式文件，等等

4 集群：多台服务器部署相同的应用构成一个集群，通过负载均衡设备共同对外提供服务

5 缓存：将数据放距离计算最近的位置加快处理速度，改善性能第一手段，可以加快访问速度，减小后端负载压力。

CDN

反向代理

本地缓存

分布式缓存Redis

6 异步：旨在系统解耦。异步架构是典型的消费者生产者模式，特性如下：

提高系统可用性

加快网站访问速度

消除并发访问高峰

7 冗余：实现高可用。数据库的冷备份和热备份

8 自动化：包括发布过程自动化，自动化代码管理，自动化测试，自动化安全检测，自动化部署，

自动化监控，自动化报警，自动化失效转移，自动化失效恢复，自动化降级，自动化分配资源

9 安全：密码，手机校验码，加密，验证码，过滤，风险控制

1 高性能

（1）性能的测试指标主要有：

响应时间：指应用执行一个操作需要的时间

并发数：指系统能够同时处理请求的数目

吞吐量：指单位时间内系统处理的请求数量

性能计数器：描述服务器或者操作系统性能的一些数据指标

（2）性能测试方法：性能测试；负载测试；压力测试；稳定性测试

（3）性能优化策略

@@@Web 前端性能优化

保护网站安全

通过配置缓存功能加速 Web 请求

实现负载均衡

减少 http 请求

使用浏览器缓存

启用压缩

AMD异步

CSS 放在页面最上面，JavaScript 放在页面最下面

减少 Cookie 传输

浏览器访问优化

异步操作

CDN 加速：本质是一个缓存，一般缓存静态资源

反向代理

@@@应用服务器性能优化：主要手段有 缓存、集群、异步

多线程(设计为无状态，使用局部对象，并发访问资源使用锁)

资源复用(单例，对象池)

数据结构

垃圾回收

分布式缓存(网站性能优化第一定律：优化考虑使用缓存优化性能)

异步操作(消息队列，削峰作用)

使用集群

代码优化

@@@存储服务器性能优化

机械硬盘 vs. 固态硬盘

B+ 树 vs. LSM 树

RAID vs. HDFS

2 高可用

目的是保证服务器硬件故障时服务依然可用、数据依然保存并能够被访问，主要手段数据和服务的冗余备份及失效转移

保证服务器硬件故障服务依然可用，数据依然保存并能够被访问。数据和服务的 ①冗余备份 ②失效转移

对于服务而言，一旦某个服务器宕机，就将服务切换到其他可用的服务器上；

对于数据而言，如果某个磁盘损坏，就从备份的磁盘（事先就做好了数据的同步复制）读取数据。

@@HTTP的无状态性：

在客户端浏览器向HTTP服务器发送请求，继而HTTP服务器将相应的资源发回给客户端这样一个过程中，无论对于客户端还是服务器，都没有必要记录这个过程，因为每一次

请求和响应都是相对独立的。一般而言，一个URL对应着唯一的超文本，而HTTP服务器也绝对公平公正，不管你是Michael，还是Jordon，它都会根据接收到的URL请求返回

相同的超文本。正是因为这样的唯一性，使得记录用户的行为状态变得毫无意义，所以，HTTP协议被设计为无状态的连接协议符合它本身的需求。

然而，随着时间的推移，人们发现静态的HTML着实无聊而乏味，增加动态生成的内容才会令Web应用程序变得更加有用。于是乎，HTML的语法在不断膨胀，其中最重要的是增加了

表单（Form）；客户端也增加了诸如脚本处理、DOM处理等功能；对于服务器，则相应的出现了CGI（Common Gateway Interface）以处理包含表单提交在内的动态请求。在这

种客户端与服务器进行动态交互的Web应用程序出现之后，HTTP无状态的特性严重阻碍了这些应用程序的实现，毕竟交互是需要承前启后的，简单的购物车程序也要知道用户到

底在之前选择了什么商品。于是，两种用于保持HTTP连接状态的技术就应运而生了，一个是Cookie，而另一个则是Session。

HTTP本身是一个无状态的连接协议，为了支持客户端与服务器之间的交互，我们就需要通过不同的技术为交互存储状态，而这些不同的技术就是Cookie和Session了。

@@应用服务的无状态性：

一个服务的确无状态，那么就能调用服务的任何方法或引用该服务的任何属性，而且只要传递相同的参数，该服务就能以相同的方式正常运行。换言之，服务中存

储的值可以被外部调用者改变，而且不会影响操作的结果。

措施

@无状态服务的失效转移

@不同机器的Session复制；

@利用负载均衡的源地址Hash算法进行session绑定，但是一旦宕机没有高可用。

@利用浏览器支持的Cookie记录Session简单易行，可用性高，但是受Cookie大小限制、每次请求响应都要传输Cookie影响性能、用户关闭了Cookie会造成访问不正常

@利用独立部署的Session服务器（Redis集群）统一管理Session，应用服务器每次读写Session时，都访问Session服务器。

@数据备份，关系数据库的热备机制就是通常所说的主从同步机制，实践中通常使用读写分离的方法来访问Master和Slave数据库，也就是说写操作只访问Master库，读操作均访问Slave库。

@失效转移：若数据服务器集群中任何一台服务器宕机，那么应用程序针对这台服务器的所有读写操作都要重新路由到其他服务器，保证数据访问不会失败。

@网站运行监控，警报系统；失效转移；自动优雅降级；用户行为日志采集；服务器性能监控；运行数据报告；监控数据采集；监控管理

3 伸缩性

伸缩性是指不需要改变网站的软硬件设计，仅通过改变部署的服务器数量就可以扩大或缩小网站的服务处理能力。

网站的伸缩性设计分为两类，一类是根据功能进行物理分离实现伸缩（比如说应用服务器，数据服务器，缓存服务区，文件服务器从一台机器分离多台；），

一类是单一功能通过集群实现伸缩（比如应用服务器进行集群，利用负载均衡分发请求。）。

前者是不同服务器部署不同的服务，提供不同的功能；后者是集群内的服务器部署相同的任务，提供相同的功能。

（1）网站架构的伸缩性设计

纵向分离(分层后分离)；横向分离(业务分割后分离)；不同功能进行物理分离实现伸缩；单一功能通过集群规模实现伸缩

（2）应用服务器集群的伸缩性设计：轮询；加权轮询；随机；最少链接；源地址散列；HTTP 重定向负载均衡；DNS 域名解析负载均衡；反向代理负载均衡；IP 负载均衡

（3）分布式缓存集群的伸缩性设计

Memcached 客户端（包括 API，路由算法，服务器列表，通信模块）

Memcached 服务器集群

Memcached 分布式缓存集群的访问模型

Memcached 分布式缓存集群的伸缩性挑战

分布式缓存的一致性 Hash 算法(一致性 Hash 环，虚拟层)

（4）数据存储服务集群的伸缩性设计

关系数据库集群的伸缩性设计

NoSQL 数据库的伸缩性设计

4 扩展性

系统架构设计层面的开闭原则，对修改关闭，对扩展开放。

设计网站可扩展架构的核心思想是模块化，并在此基础上，降低模块间的耦合性，提供模块的复用性。模块通过分布式部署，独立的模块部署在独立的服务器上（集群）从物理上分离模块之间的耦合关系。

模块分布式部署以后具体聚合方式主要有分布式消息队列和分布式服务。

（1）通过在低耦合的模块之间传输事件消息，事件驱动架构，以保持模块的松散耦合，并借助事件消息的通信完成模块间合作，典型的架构就是生产者消费者模式。在大型网站架构中，具体实现手段很多，最常用的就是分布式消息队列。

消息队列利用发布-订阅模式工作，消息发送者发布消息，一个或者多个消息接收者订阅消息。由于消息发送者不需要等待消息接受者处理数据就可以返回，系统具有更好的响应延迟；同时，在网站访问高峰，

消息可以暂时存储在消息队列中等待处理，减轻数据库等后端存储的负载压力。目前开源的和商业的分布式消息队列产品有很多，比较著名的有Apache ActiveMQ

（2）打造可复用的业务平台，使用分布式服务是降低系统耦合性的另一个重要手段。如果说分布式消息队列通过消息对象分解系统耦合性，不同子系统处理同一个消息；那么分布式服务则通过接口分解系统耦合性，不同子系统通过相同的接口描

述进行服务调用。目前国内有较多成功实施案例的开源分布式服务框架是阿里巴巴的Dubbo。

5 安全

全球大约70%的Web应用攻击都来自XSS攻击和SQL注入攻击。此外，常用的Web应用还包括CSRF、Session劫持等手段。

（1）XSS：XSS又称CSS，全称Cross SiteScript，跨站脚本攻击，是Web程序中常见的漏洞，XSS属于被动式且用于客户端的攻击方式，所以容易被忽略其危害性。其原理是攻击者向有XSS漏洞的网站中输入

(传入)恶意的HTML代码，当其它用户浏览该网站时，这段HTML代码会自动执行，从而达到攻击的目的。如，盗取用户Cookie、破坏页面结构、重定向到其它网站等。理论上，所有可输入的地方没有对输入

数据进行处理的话，都会存在XSS漏洞，漏洞的危害取决于攻击代码的威力

# 线程池&数据库连接池&多线程

Java通过Executors提供四种线程池，分别为：  
newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。  
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

**(1) newCachedThreadPool**  
创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。示例代码如下：

**Java代码**[**IMG_256**](http://cuisuqiang.iteye.com/blog/javascript:void())

1. **package** test;
2. **import** java.util.concurrent.ExecutorService;
3. **import** java.util.concurrent.Executors;
4. **public** **class** ThreadPoolExecutorTest {
5. **public** **static** **void** main(String[] args) {
6. ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
7. **for** (**int** i = 0; i < 10; i++) {
8. **final** **int** index = i;
9. **try** {
10. Thread.sleep(index \* 1000);
11. } **catch** (InterruptedException e) {
12. e.printStackTrace();
13. }
14. cachedThreadPool.execute(**new** Runnable() {
15. **public** **void** run() {
16. System.out.println(index);
17. }
18. });
19. }
20. }
21. }

线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。  
   
(2) newFixedThreadPool  
创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。示例代码如下：

**Java代码**[**IMG_257**](http://cuisuqiang.iteye.com/blog/javascript:void())

1. **package** test;
2. **import** java.util.concurrent.ExecutorService;
3. **import** java.util.concurrent.Executors;
4. **public** **class** ThreadPoolExecutorTest {
5. **public** **static** **void** main(String[] args) {
6. ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
7. **for** (**int** i = 0; i < 10; i++) {
8. **final** **int** index = i;
9. fixedThreadPool.execute(**new** Runnable() {
10. **public** **void** run() {
11. **try** {
12. System.out.println(index);
13. Thread.sleep(2000);
14. } **catch** (InterruptedException e) {
15. e.printStackTrace();
16. }
17. }
18. });
19. }
20. }
21. }

因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。  
定长线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()

(3)  newScheduledThreadPool  
创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

**Java代码**[**IMG_258**](http://cuisuqiang.iteye.com/blog/javascript:void())

1. **package** test;
2. **import** java.util.concurrent.Executors;
3. **import** java.util.concurrent.ScheduledExecutorService;
4. **import** java.util.concurrent.TimeUnit;
5. **public** **class** ThreadPoolExecutorTest {
6. **public** **static** **void** main(String[] args) {
7. ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);
8. scheduledThreadPool.schedule(**new** Runnable() {
9. **public** **void** run() {
10. System.out.println("delay 3 seconds");
11. }
12. }, 3, TimeUnit.SECONDS);
13. }
14. }

表示延迟3秒执行。

定期执行示例代码如下：

**Java代码**[**IMG_259**](http://cuisuqiang.iteye.com/blog/javascript:void())

1. **package** test;
2. **import** java.util.concurrent.Executors;
3. **import** java.util.concurrent.ScheduledExecutorService;
4. **import** java.util.concurrent.TimeUnit;
5. **public** **class** ThreadPoolExecutorTest {
6. **public** **static** **void** main(String[] args) {
7. ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);
8. scheduledThreadPool.scheduleAtFixedRate(**new** Runnable() {
9. **public** **void** run() {
10. System.out.println("delay 1 seconds, and excute every 3 seconds");
11. }
12. }, 1, 3, TimeUnit.SECONDS);
13. }
14. }

表示延迟1秒后每3秒执行一次。

(4) newSingleThreadExecutor  
创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。示例代码如下：

**Java代码**[**IMG_260**](http://cuisuqiang.iteye.com/blog/javascript:void())

1. **package** test;
2. **import** java.util.concurrent.ExecutorService;
3. **import** java.util.concurrent.Executors;
4. **public** **class** ThreadPoolExecutorTest {
5. **public** **static** **void** main(String[] args) {
6. ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
7. **for** (**int** i = 0; i < 10; i++) {
8. **final** **int** index = i;
9. singleThreadExecutor.execute(**new** Runnable() {
10. **public** **void** run() {
11. **try** {
12. System.out.println(index);
13. Thread.sleep(2000);
14. } **catch** (InterruptedException e) {
15. e.printStackTrace();
16. }
17. }
18. });
19. }
20. }
21. }

结果依次输出，相当于顺序执行各个任务。

你可以使用JDK自带的监控工具来监控我们创建的线程数量，运行一个不终止的线程，创建指定量的线程，来观察：  
工具目录：**C:\Program Files\Java\jdk1.6.0\_06\bin\jconsole.exe**  
运行程序做稍微修改：

**Java代码**[**IMG_261**](http://cuisuqiang.iteye.com/blog/javascript:void())

1. **package** test;
2. **import** java.util.concurrent.ExecutorService;
3. **import** java.util.concurrent.Executors;
4. **public** **class** ThreadPoolExecutorTest {
5. **public** **static** **void** main(String[] args) {
6. ExecutorService singleThreadExecutor = Executors.newCachedThreadPool();
7. **for** (**int** i = 0; i < 100; i++) {
8. **final** **int** index = i;
9. singleThreadExecutor.execute(**new** Runnable() {
10. **public** **void** run() {
11. **try** {
12. **while**(**true**) {
13. System.out.println(index);
14. Thread.sleep(10 \* 1000);
15. }
16. } **catch** (InterruptedException e) {
17. e.printStackTrace();
18. }
19. }
20. });
21. **try** {
22. Thread.sleep(500);
23. } **catch** (InterruptedException e) {
24. e.printStackTrace();
25. }
26. }
27. }
28. }
29. 为什么要使用线程池  
    在java中，如果每个请求到达就创建一个新线程，开销是相当大的。在实际使用中，服务器在创建和销毁线程上花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用户请求的时间和资源要多的多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个jvm里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，服务器应用程序需要采取一些办法来限制任何给定时刻处理的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务，这就是“池化资源”技术产生的原因。  
     线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重复使用线程，线程创建的开销就被分摊到了多个任务上了，而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使用应用程序响应更快。另外，通过适当的调整线程中的线程数目可以防止出现资源不足的情况。  
    2.线程池的组成部分  
     一个比较简单的线程池至少应包含线程池管理器、工作线程、任务列队、任务接口等部分。其中线程池管理器的作用是创建、销毁并管理线程池，将工作线程放入线程池中；工作线程是一个可以循环执行任务的线程，在没有任务是进行等待；任务列队的作用是提供一种缓冲机制，将没有处理的任务放在任务列队中；任务接口是每个任务必须实现的接口，主要用来规定任务的入口、任务执行完后的收尾工作、任务的执行状态等，工作线程通过该接口调度任务的执行。  
       线程池管理器至少有下列功能：创建线程池，销毁线程池，添加新任务。  
     工作线程是一个可以循环执行任务的线程，在没有任务时将等待。  
     任务接口是为所有任务提供统一的接口，以便工作线程处理。任务接口主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等。  
    3.线程池适合应用的场合  
     当一个服务器接受到大量短小线程的请求时，使用线程池技术是非常合适的，它可以大大减少线程的创建和销毁次数，提高服务器的工作效率。但是线程要求的运动时间比较长，即线程的运行时间比

**数据库连接池：**

1. 基本原理：在内部对象池中，维护一定数量的[数据库](http://lib.csdn.net/base/mysql)连接，并对外暴露数据库连接的获取和返回方法。

如外部使用者可通过getConnection方法获取数据库连接，使用完毕后再通过releaseConnection方法将连接返回，注意此时的连接并没有关闭，而是由连接池管理器回收，并为下一次使用做好准备。

2.作用

   ①资源重用

      由于数据库连接得到重用，避免了频繁创建、释放连接引起的大量性能开销。在减少系统消耗的基础上，增进了系统环境的平稳性（减少内存碎片以级数据库临时进程、线程的数量）

   ②更快的系统响应速度

      数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于池内备用。此时连接池的初始化操作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而缩减了系统整体响应时间。

   ③新的资源分配手段

      对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接的配置，实现数据库连接技术。

   ④统一的连接管理，避免数据库连接泄露

     在较为完备的数据库连接池实现中，可根据预先的连接占用超时设定，强制收回被占用的连接，从而避免了常规数据库连接操作中可能出现的资源泄露

**替换目标:替换掉C3P0,用druid来替换**

替换原因:

1、性能方面 hikariCP>druid>tomcat-jdbc>dbcp>c3p0 。hikariCP的高性能得益于最大限度的避免锁竞争。

2、druid功能最为全面，sql拦截等功能，统计数据较为全面，具有良好的扩展性。

3、综合性能，扩展性等方面，可考虑使用druid或者hikariCP连接池，比较方便对jdbc接口进行监控跟踪等。

4、可开启prepareStatement缓存，对性能会有大概20%的提升。

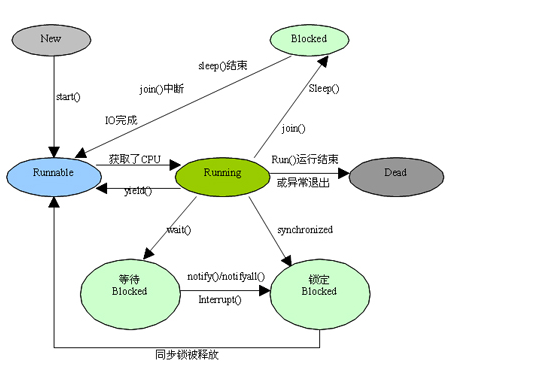
psCache是connection私有的，所以不存在线程竞争的问题，开启pscache不会存在竞争的性能损耗。

psCache的key为prepare执行的sql和catalog等，value对应的为prepareStatement对象。开启缓存主要是减少了解析sql的开销。

5、3p0历史悠久，代码及其复杂，不利于维护。并且存在deadlock的潜在风险。

6、Druid可以打印SQL，慢查询方面的日志

# 多线程



**Java提供的线程同步方式**

**1、synchronized关键字**

方法或代码块的互斥性来完成实际上的一个原子操作。（方法或代码块在被一个线程调用时，其他线程处于等待状态）

所有的Java对象都有一个与synchronzied关联的监视器对象（monitor），允许线程在该监视器对象上进行加锁和解锁操作。

a、静态方法：Java类对应的Class类的对象所关联的监视器对象。

b、实例方法：当前对象实例所关联的监视器对象。

c、代码块：代码块声明中的对象所关联的监视器对象。

注：当锁被释放，对共享变量的修改会写入主存；当活得锁，CPU缓存中的内容被置为无效。编译器在处理synchronized方法或代码块，不会把其中包含的代码移动到synchronized方法或代码块之外，从而避免了由于代码重排而造成的问题。

例：以下方法getNext()和getNextV2() 都获得了当前实例所关联的监视器对象

**[java]** [view plain](http://blog.csdn.net/escaflone/article/details/10418651) [copy](http://blog.csdn.net/escaflone/article/details/10418651)

1. **public** **class** SynchronizedIdGenerator{
2. **private** **int** value = 0;
3. **public** **synchronized** **int** getNext(){
4. **return** value++;
5. }
6. **public** **int** getNextV2(){
7. **synchronized**(**this**){
8. **return** value++;
9. }
10. }
11. }

**2、Object类的wait、notify和notifyAll方法**

生产者和消费者模式，判断缓冲区是否满来消费，缓冲区是否空来生产的逻辑。如果用while 和 volatile也可以做，不过本质上会让线程处于忙等待，占用CPU时间，对性能造成影响。

wait： 将当前线程放入，该对象的等待池中，线程A调用了B对象的wait()方法，线程A进入B对象的等待池，并且释放B的锁。（这里，线程A必须持有B的锁，所以调用的代码必须在synchronized修饰下，否则直接抛出java.lang.IllegalMonitorStateException异常）。

notify：将该对象中等待池中的线程，随机选取一个放入对象的锁池，当当前线程结束后释放掉锁， 锁池中的线程即可竞争对象的锁来获得执行机会。

notifyAll：将对象中等待池中的线程，全部放入锁池。

（notify锁唤醒的线程选择由虚拟机实现来决定，不能保证一个对象锁关联的等待集合中的线程按照所期望的顺序被唤醒，很可能一个线程被唤醒之后，发现他所要求的条件并没有满足，而重新进入等待池。因为当等待池中包含多个线程时，一般使用notifyAll方法，不过该方法会导致线程在没有必要的情况下被唤醒，之后又马上进入等待池，对性能有影响，不过能保证程序的正确性）

工作流程：

a、Consumer线程A 来 看产品，发现产品为空，调用产品对象的wait()，线程A进入产品对象的等待池并释放产品的锁。

b、Producer线程B获得产品的锁，执行产品的notifyAll()，Consumer线程A从产品的等待池进入锁池，Producer线程B生产产品，然后退出释放锁。

c、Consumer线程A获得产品锁，进入执行，发现有产品，消费产品，然后退出。

例子：

**[java]** [view plain](http://blog.csdn.net/escaflone/article/details/10418651) [copy](http://blog.csdn.net/escaflone/article/details/10418651)

1. **public** **synchronized** String pop(){
2. **this**.notifyAll();// 唤醒对象等待池中的所有线程，可能唤醒的就是 生产者（当生产者发现产品满，就会进入对象的等待池，这里代码省略，基本略同）
3. **while**(index == -1){//如果发现没产品，就释放锁，进入对象等待池
4. **this**.wait();
5. }//当生产者生产完后，消费者从this.wait()方法再开始执行，第一次还会执行循环，万一产品还是为空，则再等待，所以这里必须用while循环，不能用if
6. String good = buffer[index];
7. buffer[index] = **null**;
8. index--;
9. **return** good;// 消费完产品，退出。
10. }

注：wait()方法有超时和不超时之分，超时的在经过一段时间，线程还在对象的等待池中，那么线程也会推出等待状态。

**3、线程状态转换：**

已经废弃的方法：stop、suspend、resume、destroy，这些方法在实现上时不安全的。

线程的状态：NEW、RUNNABLE、BLOCKED、WAITING、TIMED\_WAITING（有超时的等待）、TERMINATED。

a、方法sleep()进入的阻塞状态，不会释放对象的锁（即大家一起睡，谁也别想执行代码），所以不要让sleep方法处在synchronized方法或代码块中，否则造成其他等待获取锁的线程长时间处于等待。

b、方法join()则是主线程等待子线程完成，再往下执行。例如main方法新建两个线程A和B

**[java]** [view plain](http://blog.csdn.net/escaflone/article/details/10418651) [copy](http://blog.csdn.net/escaflone/article/details/10418651)

1. **public** **static** **void** main(String[] args) **throws** InterruptedException {
2. Thread t1 = **new** Thread(**new** ThreadTesterA());
3. Thread t2 = **new** Thread(**new** ThreadTesterB());
4. t1.start();
5. t1.join(); // 等t1执行完再往下执行
6. t2.start();
7. t2.join(); // 在虚拟机执行中，这句可能被忽略
8. }

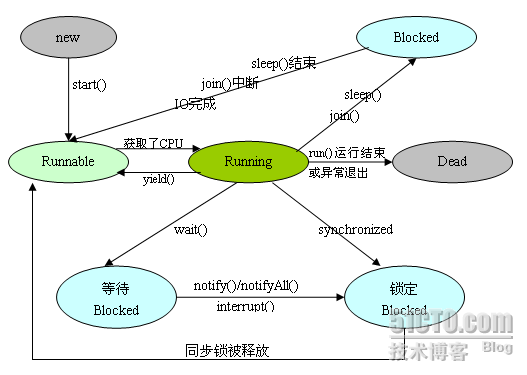
c、方法interrupt()，向被调用的对象线程发起中断请求。如线程A通过调用线程B的d的interrupt方法来发出中断请求，线程B来处理这个请求，当然也可以忽略，这不是必须的。Object类的wait()、Thread类的join()和sleep方法都会抛出受检异常java.lang.InterruptedException，通过interrupt方法中断该线程会导致线程离开等待状态。对于wait()调用来说，线程需要重新获取监视器对象上的锁之后才能抛出InterruptedException异常，并致以异常的处理逻辑。

可以通过Thread类的isInterrupted方法来判断是否有中断请求发生，通常可以利用这个方法来判断是否退出线程（类似上面的volatitle修饰符的例子）；

Thread类还有个方法Interrupted()，该方法不但可以判断当前线程是否被中断，还会清楚线程内部的中断标记，如果返回true，即曾被请求中断，同时调用完后，清除中断标记。

如果一个线程在某个对象的等待池，那么notify和interrupt 都可以使该线程从等待池中被移除。如果同时发生，那么看实际发生顺序。如果是notify先，那照常唤醒，没影响。如果是interrupt先，并且虚拟机选择让该线程中断，那么即使nofity，也会忽略该线程，而唤醒等待池中的另一个线程。

e、yield()，尝试让出所占有的CPU资源，让其他线程获取运行机会，对[**操作系统**](http://lib.csdn.net/base/operatingsystem)上的调度器来说是一个信号，不一定立即切换线程。（在实际开发中，[**测试**](http://lib.csdn.net/base/softwaretest)阶段频繁调用yeid方法使线程切换更频繁，从而让一些多线程相关的错误更容易暴露出来）。



**四、非阻塞方式**

线程之间同步机制的核心是监视对象上的锁，竞争锁来获得执行代码的机会。当一个对象获取对象的锁，然后其他尝试获取锁的对象会处于等待状态，这种锁机制的实现方式很大程度限制了多线程程序的吞吐量和性能（线程阻塞），且会带来死锁（线程A有a对象锁，等着获取b对象锁，线程B有b对象锁，等待获取a对象锁）和优先级倒置（优先级低的线程获得锁，优先级高的只能等待对方释放锁）等问题。

如果能不阻塞线程，又能保证多线程程序的正确性，就能有更好的性能。

在程序中，对共享变量的使用一般遵循一定的模式，即读取、修改和写入三步组成。之前碰到的问题是，这三步执行中可能线程执行切换，造成非原子操作。锁机制是把这三步变成一个原子操作。

目前CPU本身实现 将这三步 合起来 形成一个原子操作，无需线程锁机制干预，常见的指令是“比较和替换”（compare and swap,CAS），这个指令会先比较某个内存地址的当前值是不是指定的旧指，如果是，就用新值替换，否则什么也不做，指令返回的结果是内存地址的当前值。通过CAS指令可以实现不依赖锁机制的非阻塞[**算法**](http://lib.csdn.net/base/datastructure)。一般做法是把CAS指令的调用放在一个无限循环中，不断尝试，知道CAS指令成功完成修改。

java.util.concurrent.atomic包中提供了CAS指令。（不是所有CPU都支持CAS，在某些平台，java.util.concurrent.atomic的实现仍然是锁机制）

**atomic包中提供的Java类分成三类：**

**1、支持以原子操作来进行更新的数据类型的Java类**（AtomicBoolean、AtomicInteger、AtomicReference），在内存模型相关的语义上，这四个类的对象类似于volatile变量。

类中的常用方法：

a、compareAndSet：接受两个参数，一个是期望的旧值，一个是替换的新值。

b、weakCompareAndSet：效果同compareAndSet（JSR中表示weak原子方式读取和有条件地写入变量但不创建任何 happen-before 排序，但在源代码中和compareAndSet完全一样，所以并没有按JSR实现）

c、get和set：分别用来直接获取和设置变量的值。

d、lazySet：与set类似，但允许编译器把lazySet方法的调用与后面的指令进行重排，因此对值得设置操作有可能被推迟。

例：

**[java]** [view plain](http://blog.csdn.net/escaflone/article/details/10418651) [copy](http://blog.csdn.net/escaflone/article/details/10418651)

1. **public** **class** AtomicIdGenerator{
2. **private** **final** AtomicInter counter = **new** AtomicInteger(0);
3. **public** **int** getNext(){
4. **return** counter.getAndIncrement();
5. }
6. }
7. // getAndIncrement方法的内部实现方式，这也是CAS方法的一般模式，CAS方法不一定成功，所以包装在一个无限循环中，直到成功
8. **public** **final** **int** getAndIncrement(){
9. **for**(;;){
10. **int** current = get();
11. **int** next = current +1;
12. **if**(compareAndSet(current,next))
13. **return** current;
14. }
15. }

2、提供对数组类型的变量进行处理的Java类，AtomicIntegerArray、AtomicLongArray和AtomicReferenceArray类。（同上，只是放在类数组里，调用时也只是多了一个操作元素索引的参数）

**3、通过反射的方式对任何对象中包含的volatitle变量使用CAS方法**，AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater。他们提供了一种方式把CAS的功能扩展到了任何Java类中声明为volatitle的域上。（灵活，但语义较弱，因为对象的volatitle可能被非atomic的其他方式被修改）

**[java]** [view plain](http://blog.csdn.net/escaflone/article/details/10418651) [copy](http://blog.csdn.net/escaflone/article/details/10418651)

1. **public** **class** TreeNode{
2. **private** **volatile** TreeNode parent;
3. // 静态工厂方法
4. **private** **static** **final** AtomicReferenceFieldUpdater<TreeNode, TreeNode> parentUpdater = AtomicReferenceFieldUpdater.newUpdater(TreeNode.**class**,TreeNode.**class**,"parent");
5. **public** **boolean** compareAndSetParent(TreeNode expect, TreeNode update){
6. **return** parentUpdater.compareAndSet(**this**, expect, update);
7. }
8. }

注：java.util.concurrent.atomic包中的Java类属于比较底层的实现，一般作为java.util.concurrent包中很多非阻塞的数据结构的实现基础。

比较多的用AtomicBoolean、AtomicInteger、AtomicLong和AtomicReference。在实现线程安全的计数器时，AtomicInteger和AtomicLong类时最佳的选择。

**五、高级同步机制（比synchronized更灵活的加锁机制）**

synchronized和volatile，以及wait、notify等方法抽象层次低，在程序开发中使用比较繁琐，易出错。

而多线程之间的交互来说，存在某些固定的模式，如生产者-消费者和读者-写者模式，把这些模式抽象成高层API，使用起来会非常方便。

java.util.concurrent包为多线程提供了高层的API，满足日常开发中的常见需求。

常用接口

**1、Lock接口，表示一个锁方法：**

a、lock()，获取所，如果无法获取所锁，会处于等待状态

b、unlock()，释放锁。（一般放在finally代码块中）

c、lockInterruptibly()，与lock()类似，但允许当前线程在等待获取锁的过程中被中断。（所以要处理InterruptedException）

d、tryLock()，以非阻塞方式获取锁，如果无法获取锁，则返回false。（tryLock()的另一个重载可以指定超时，如果指定超时，当无法获取锁，会等待而阻塞，同时线程可以被中断）

**2、ReadWriteLock接口，表示两个锁，读取的共享锁和写入的排他锁。（适合常见的读者--写者场景）**

ReadWriteLock接口的readLock和writeLock方法来获取对应的锁的Lock接口的实现。

在多数线程读取，少数线程写入的情况下，可以提高多线程的性能，提高使用该[**数据结构**](http://lib.csdn.net/base/datastructure)的吞吐量。

如果是相反的情况，较多的线程写入，则接口会降低性能。

**3、ReentrantLock类和ReentrantReadWriteLock，分别为上面两个接口的实现类。**

他们具有重入性：即允许一个线程多次获取同一个锁（他们会记住上次获取锁并且未释放的线程对象，和加锁的次数，getHoldCount()）

同一个线程每次获取锁，加锁数+1，每次释放锁，加锁数-1，到0，则该锁被释放，可以被其他线程获取。

**[java]** [view plain](http://blog.csdn.net/escaflone/article/details/10418651) [copy](http://blog.csdn.net/escaflone/article/details/10418651)

1. **public** **class** LockIdGenrator{
2. //new ReentrantLock(true)是重载，使用更加公平的加锁机制，在锁被释放后，会优先给等待时间最长的线程，避免一些线程长期无法获得锁
3. **private** **int** ReentrantLock lock = ReentrantLock();
4. privafte **int** value = 0;
5. **public** **int** getNext(){
6. lock.lock();      //进来就加锁，没有锁会等待
7. **try**{
8. **return** value++;//实际操作
9. }**finally**{
10. lock.unlock();//释放锁
11. }
12. }
13. }

注：重入性减少了锁在各个线程之间的等待，例如便利一个HashMap，每次next()之前加锁，之后释放，可以保证一个线程一口气完成便利，而不会每次next()之后释放锁，然后和其他线程竞争，降低了加锁的代价， 提供了程序整体的吞吐量。（即，让一个线程一口气完成任务，再把锁传递给其他线程）。  
4、Condition接口，Lock接口代替了synchronized，Condition接口替代了object的wait、nofity。

a、await()，使当前线程进入等待状态，知道被唤醒或中断。重载形式可以指定超时时间。

b、awaitNanos()，以纳秒为单位等待。

c、awaitUntil()，指定超时发生的时间点，而不是经过的时间，参数为java.util.Date。

d、awaitUninterruptibly()，前面几种会响应其他线程发出的中断请求，他会无视，直到被唤醒。

注：与Object类的wait()相同，await()会释放其所持有的锁。

e、signal()和signalAll， 相当于 notify和notifyAll

**[java]** [view plain](http://blog.csdn.net/escaflone/article/details/10418651) [copy](http://blog.csdn.net/escaflone/article/details/10418651)

1. Lock lock = **new** ReentrantLock();
2. Condition condition = lock.newCondition();
3. lock.lock();
4. **try**{
5. **while**(/\*逻辑条件不满足\*/){
6. condition.await();
7. }
8. }**finally**{
9. lock.unlock();
10. }

**九、多线程任务的执行**

过去线程的执行，是先创建Thread类的想，再调用start方法启动，这种做法要求开发人员对线程进行维护，在线程较多时，一般创建一个线程池同一管理，同时降低重复创建线程的开销

在J2SE5.0中，java.util.concurrent包提供了丰富的用来管理线程和执行任务的实现。

**1、基本接口（描述任务）**

a、Callable接口：

Runnable接口受限于run方法的类型签名，而Callable只有一个方法call()，可以有返回值，可以抛出受检异常。

b、Future接口：

过去，需要异步线程的任务执行结果，要求主线程和任务执行线程之间进行同步和数据传递。

Future简化了任务的异步执行，作为异步操作的一个抽象。调用get()方法可以获取异步的执行结果，如果任务没有执行完，会等待，直到任务完成或被取消，cancel()可以取消。

c、Delayed接口：

延迟执行任务，getDelay()返回当前剩余的延迟时间，如果不大于0，说明延迟时间已经过去，应该调度并执行该任务。

**2、组合接口（描述任务）**

a、RunnableFuture接口：继承自Runnable接口和Future接口。

当来自Runnalbe接口中的run方法成功执行之后，相当于Future接口表示的异步任务已经完成，可以通过get()获取运行结果。

b、ScheduledFuture接口：继承Future接口和Delayed接口，表示一个可以调用的异步操作。

c、RunnableScheduledFuture接口：继承自Runnable、Delayed和Future，接口中包含isPeriodic，表明该异步操作是否可以被重复执行。

**3、Executor接口、ExcutorServer接口、ScheduleExecutorService接口和CompletionService接口（描述任务执行）**

a、executor接口，execute()用来执行一个Runnable接口的实现对象，不同的Executor实现采取不同执行策略，但提供的任务执行功能比较弱。

b、excutorServer接口，继承自executor；

提供了对任务的管理：submit()，可以吧Callable和Runnable作为任务提交，得到一个Future作为返回，可以获取任务结果或取消任务。

提供批量执行：invokeAll()和invokeAny()，同时提交多个Callable；invokeAll()，会等待所有任务都执行完成，返回一个包含每个任务对应Future的列表；invokeAny()，任何一个任务成功完成，即返回该任务结果。

提供任务关闭：shutdown()、shutdownNow()来关闭服务，前者不允许新的任务提交，后者试图终止正在运行和等待的任务，并返回已经提交单没有被运行的任务列表。（两个方法都不会等待服务真正关闭，只是发出关闭请求。）。shutdownDow，通常做法是向线程发出中断请求，所以确保提交的任务实现了正确的中断处理逻辑。

c、ScheduleExecutorService接口，继承自excutorServer接口：支持任务的延迟执行和定期执行，可以执行Callable或Runnable。

schedule()，调度一个任务在延迟若干时间之后执行；

scheduleAtFixedRate()：在初始延迟后，每隔一段时间循环执行；在下一次执行开始时，上一次执行可能还未结束。（同一时间，可能有多个）

scheduleWithFixedDelay：同上，只是在上一次任务执行完后，经过给定的间隔时间再开始下一次执行。（同一时间，只有一个）

以上三个方法都返回ScheduledFuture接口的实现对象。

d、CompletionService接口，共享任务执行结果。

通常在使用ExecutorService接口，通过submit提交任务，并得到一个Future接口来获取任务结果，如果任务提交者和执行结果的使用者是程序的不同部分，那就要把Future在不同部分进行传递；而CompletionService就是解决这个问题，程序不同部分可以共享CompletionService，任务提交后，执行结果可以通过take（阻塞），poll（非阻塞）来获取。

标准库提供的实现是 ExecutorCompletionService，在创建时，需要提供一个Executor接口的实现作为参数，用来实际执行任务。

例：多线程方式下载文件

**线程的几个方法：**join()：等待此线程死亡后再继续，可使异步线程变为同步线程  
interrupt()：中断线程，被中断线程会抛InterruptedException  
  
线程通信：wait(),notify()   - 典型应用：放取鸡蛋  
wait() 等待获取锁：

表示等待获取某个锁

执行了该方法的线程释放对象的锁，JVM会把该线程放到对象的等待池中。该线程等待其它线程唤醒   
notify() 执行该方法的线程唤醒在对象的等待池中等待的一个线程，JVM从对象的等待池中随机选择一个线程，把它转到对象的锁池中。使线程由阻塞队列进入就绪状态  
  
sleep()：让当前正在执行的线程休眠，有一个用法可以代替yield函数——sleep(0)  
yield()：暂停当前正在执行的线程对象，并执行其他线程。也就是交出CPU一段时间  
  
sleep和yield区别：  
1、sleep()方法会给其他线程运行的机会，而不考虑其他线程的优先级，因此会给较低线程一个运行的机会；yield()方法只会给相同优先级或者更高优先级的线程一个运行的机会。   
2、当线程执行了sleep(long millis)方法后，将转到阻塞状态，参数millis指定睡眠时间；当线程执行了yield()方法后，将转到就绪状态。   
3、sleep()方法声明抛出InterruptedException异常，而yield()方法没有声明抛出任何异常   
4、sleep()方法比yield()方法具有更好的移植性   
  
如果希望明确地让一个线程给另外一个线程运行的机会，可以采取以下的办法之一：  
1、 调整各个线程的优先级   
2、 让处于运行状态的线程调用Thread.sleep()方法   
3、 让处于运行状态的线程调用Thread.yield()方法   
4、 让处于运行状态的线程调用另一个线程的join()方法

首先，**wait()和notify(),notifyAll()是Object类的方法，sleep()和yield()是Thread类的方法**。  
  
(1).常用的wait方法有**wait()**和**wait(long** **timeout)**:  
    void wait() 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。   
    void wait(long timeout) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。  
    **wait()后，线程会释放掉它所占有的“锁标志”**，从而使线程所在对象中的其它synchronized数据可被别的线程使用。  
    wait()和notify()因为会对对象的“锁标志”进行操作，所以它们必须在synchronized函数或synchronized　 block中进行调用。如果在non-synchronized函数或non-synchronized　block中进行调用，虽然能编译通过，但在运 行时会发生IllegalMonitorStateException的异常。  
  
(2).**Thread.sleep(long millis),必须带有一个时间参数**。  
    sleep(long)使当前线程进入停滞状态，所以执行sleep()的线程在指定的时间内肯定不会被执行；  
    sleep(long)可使优先级低的线程得到执行的机会，当然也可以让同优先级和高优先级的线程有执行的机会；  
    sleep(long)是不会释放锁标志的。  
  
(3).**yield()没有参数。**  
    sleep 方法使当前运行中的线程睡眼一段时间，进入不可运行状态，这段时间的长短是由程序设定的，yield 方法使当前线程让出CPU占有权，但让出的时间是不可设定的。  
    **yield()也不会释放锁标志。**  
  
    实际上，yield()方法对应了如下操作： 先检测当前是否有相同优先级的线程处于同可运行状态，如有，则把 CPU 的占有权交给此线程，否则继续运行原来的线程。所以yield()方法称为“退让”，它把运行机会让给了同等优先级的其他线程。  
  
    sleep方法允许较低优先级的线程获得运行机会，但yield()方法执行时，当前线程仍处在可运行状态，所以不可能让出较低优先级的线程些时获得CPU占有权。 在一个运行系统中，如果较高优先级的线程没有调用 sleep 方法，又没有受到 I/O阻塞，那么较低优先级线程只能等待所有较高优先级的线程运行结束，才有机会运行。  
  
    yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行。所以yield()只能使同优先级的线程有执行的机会。

**volitile 语义：**volatile相当于synchronized的弱实现，也就是说volatile实现了类似synchronized的语义，却又没有锁机制。它确保对volatile字段的更新以可预见的方式告知其他的线程。  
volatile包含以下语义：  
（1）Java 存储模型不会对valatile指令的操作进行重排序：这个保证对volatile变量的操作时按照指令的出现顺序执行的。  
（2）volatile变量不会被缓存在寄存器中（只有拥有线程可见）或者其他对CPU不可见的地方，每次总是从主存中读取volatile变量的结果。也就是说对于volatile变量的修改，其它线程总是可见的，并且不是使用自己线程栈内部的变量。也就是在happens-before法则中，对一个valatile变量的写操作后，其后的任何读操作理解可见此写操作的结果。  
尽管volatile变量的特性不错，但是volatile并不能保证线程安全的，也就是说volatile字段的操作不是原子性的，volatile变量只能保证可见性（一个线程修改后其它线程能够理解看到此变化后的结果），要想保证原子性，目前为止只能加锁！  
  
**数据同步：**

线程同步的特征：   
1、 如果一个同步代码块和非同步代码块同时操作共享资源，仍然会造成对共享资源的竞争。因为当一个线程执行一个对象的同步代码块时，其他的线程仍然可以执行对象的非同步代码块。（所谓的线程之间保持同步，是指不同的线程在执行同一个对象的同步代码块时，因为要获得对象的同步锁而互相牵制）   
2、 每个对象都有唯一的同步锁   
3、 在静态方法前面可以使用synchronized修饰符。   
4、 当一个线程开始执行同步代码块时，并不意味着必须以不间断的方式运行，进入同步代码块的线程可以执行Thread.sleep()或者执行Thread.yield()方法，此时它并不释放对象锁，只是把运行的机会让给其他的线程。   
5、 Synchronized声明不会被继承，如果一个用synchronized修饰的方法被子类覆盖，那么子类中这个方法不在保持同步，除非用synchronized修饰。

# 分布式架构

分布式——一个高大上的名词，是计算机软件设计中人民群众喜闻乐见的“逼格满满”、“不明觉厉”的几个名词之一。但很可惜，这玩意儿一点也不复杂，甚至有些“简单”。不信？你只要遵循下述步骤即可将任何一个软件拆分为“分布式”的：

1. 将你的整个软件视为一个系统（不管它有多复杂）
2. 将整个系统分割为一系列的 Process（进程）， 每个 Process 完成一定的功能
3. 将这些 Process 分散到不同的机器上。分散后，选择若干种（没错一种可能不够）通信协议把他们连接起来

* 为了性能扩展——系统负载高，单台机器无法承载，希望通过使用多台机器来提高系统的负载能力
* 为了增强可靠性——软件不是完美的，网络不是完美的，甚至机器本身也不可能是完美的，随时可能会出错，为了避免故障，需要将业务分散开保留一定的冗余度

在以提供 Service 为主的服务端软件开发过程中常常遇到这些问题。

# 负载均衡算法

* Dubbo 负载均衡策略提供下列四种方式：
* Random LoadBalance 随机，按权重设置随机概率。 Dubbo的默认负载均衡策略  
  在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。
* RoundRobin LoadBalance 轮循，按公约后的权重设置轮循比率。  
  存在慢的提供者累积请求问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。
* LeastActive LoadBalance 最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。  
  使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。
* ConsistentHash LoadBalance 一致性Hash，相同参数的请求总是发到同一提供者。  
  当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

3．1 轮转法：

    轮转算法是所有调度算法中最简单也最容易实现的一种方法。在一个任务队列里，队列的每个成员（节点）都具有相同的地位，轮转法简单的在这组成员中顺序轮转选择。在负载平衡环境中，均衡器将新的请求轮流发给节点队列中的下一节点，如此连续、周而复始，每个集群的节点都在相等的地位下被轮流选择。这个算法在DNS域名轮询中被广泛使用。

    轮转法的活动是可预知的，每个节点被选择的机会是1/N，因此很容易计算出节点的负载分布。轮转法典型的适用于集群中所有节点的处理能力和性能均相同的情况，在实际应用中，一般将它与其他简单方法联合使用时比较有效。

3．2 散列法

    散列法也叫哈希法（HASH），通过单射不可逆的HASH函数，按照某种规则将网络请求发往集群节点。哈希法在其他几类平衡算法不是很有效时会显示出特别的威力。例如，在前面提到的UDP会话的情况下，由于轮转法和其他几类基于连接信息的算法，无法识别出会话的起止标记，会引起应用混乱。

    而采取基于数据包源地址的哈希映射可以在一定程度上解决这个问题：将具有相同源地址的数据包发给同一服务器节点，这使得基于高层会话的事务可以以适当的方式运行。相对称的是，基于目的地址的哈希调度算法可以用在Web Cache集群中，指向同一个目标站点的访问请求都被负载平衡器发送到同一个Cache服务节点上，以避免页面缺失而带来的更新Cache问题。

3．3 最少连接法

    在最少连接法中，平衡器纪录目前所有活跃连接，把下一个新的请求发给当前含有最少连接数的节点。这种算法针对TCP连接进行，但由于不同应用对系统资源的消耗可能差异很大，而连接数无法反映出真实的应用负载，因此在使用重型Web服务器作为集群节点服务时（例如Apache服务器），该算法在平衡负载的效果上要打个折扣。为了减少这个不利的影响，可以对每个节点设置最大的连接数上限（通过阈值设定体现）。

3．4 最低缺失法

    在最低缺失法中，平衡器长期纪录到各节点的请求情况，把下个请求发给历史上处理请求最少的节点。与最少连接法不同的是，最低缺失记录过去的连接数而不是当前的连接数。

3．5 最快响应法

    平衡器记录自身到每一个集群节点的网络响应时间，并将下一个到达的连接请求分配给响应时间最短的节点，这种方法要求使用ICMP包或基于UDP包的专用技术来主动探测各节点。

    在大多数基于LAN的集群中，最快响应算法工作的并不是很好，因为LAN中的ICMP包基本上都在10ms内完成回应，体现不出节点之间的差异；如果在 WAN上进行平衡的话，响应时间对于用户就近选择服务器而言还是具有现实意义的；而且集群的拓扑越分散这种方法越能体现出效果来。这种方法是高级平衡基于拓扑结构重定向用到的主要方法。

3．6 加权法

    加权方法只能与其他方法合用，是它们的一个很好的补充。加权算法根据节点的优先级或当前的负载状况（即权值）来构成负载平衡的多优先级队列，队列中的每个等待处理的连接都具有相同处理等级，这样在同一个队列里可以按照前面的轮转法或者最少连接法进行均衡，而队列之间按照优先级的先后顺序进行均衡处理。在这里权值是基于各节点能力的一个估计值。

# 本地事务和分布式事务

1：本地事务

　　轻量级事务管理器，内核事务管理器都只支持本地事务。本地事务相对简单，这儿不作重点简述。

2：分布式事务

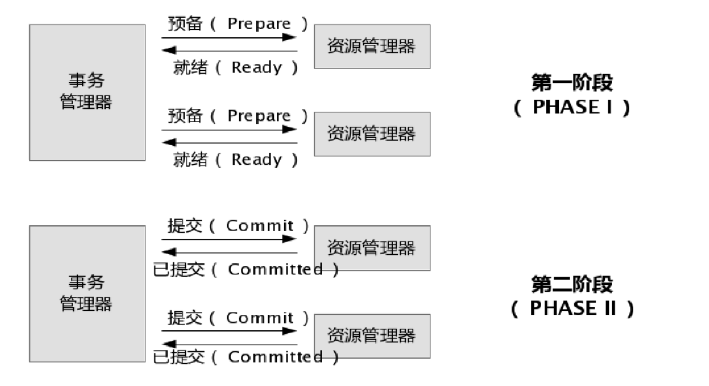
　　理解分布式事务是怎样实现的，事务提交树是关键。

　　事务提交树：事务提交树的根是事务初始化服务所在的机器的DTC，它在整个事务提交过程中充当着总协调者，又被称为全局提交协调器。资源管理器充当着事务提交树的叶子节点，它们的父结点为本机的DTC，分布于不同机器的DTC按照事务的传播路径形成了上下级关系。

1、单数据库事务

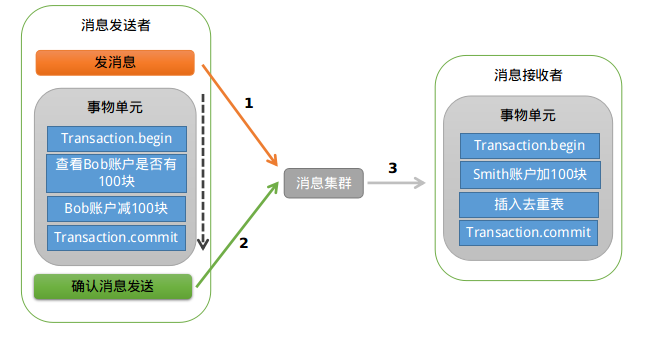
      如果应用系统是单一的数据库，那么这个很好保证，利用数据库的事务特性来满足事务的一致性，这时候的一致性是强一致性的。对于java应用系统来讲，很少直接通过事务的start和commit以及rollback来硬编码，大多通过spring的事务模板或者声明式事务来保证；

1. 多数据库事务



     针对多数据库事务可以根据二阶段提交协议，采用spring 3.0 + Atomikos + JTA进行支持；缺点是~两阶段提交涉及到多个节点的网络通信,通信时间如果过长,事务的相对时间也就会过长,那么锁定资源的时间也就长了.在高并发的服务中,就会存在严重的性能瓶劲

1. 基于**事务型消息队列**的最终一致性



     借助消息队列，在处理业务逻辑的地方发送消息，业务逻辑处理成功后，提交消息，确保消息是发送成功的，之后消息队列投递来进行处理，如果成功，则结束，如果没有成功，则重试，直到成功，不过仅仅适用业务逻辑中，第一阶段成功，第二阶段必须成功的场景。对应上图中的C流程。

4、基于**消息队列+定时补偿机制**的最终一致性

     前面部分和上面基于事务型消息的队列，不同的是，第二阶段重试的地方，不再是消息中间件自身的重试逻辑了，而是单独的补偿任务机制。其实在大多数的逻辑中，第二阶段失败的概率比较小，所以单独独立补偿任务表出来，可以更加清晰，能够比较明确的直到当前多少任务是失败的。对应上图的E流程。

5、异步回调机制的引入

     A应用调用B，在同步调用的返回结果中，B返回成功给到A，一般情况下，这时候就结束了，其实在99.99%的情况是没问题的，但是有时候为了确保100%，记住最起码在系统设计中100%，这时候B系统再回调A一下，告诉A，你调用我的逻辑，确实成功了。其实这个逻辑，非常类似TCP协议中的三次握手。上图中的B流程。

6、类似double check机制的确认机制

    还是上图中异步回调的过程，A在同步调用B，B返回成功了。这次调用结束了，但是A为了确保，在过一段时间，这个时间可以是几秒，也可以是每天定时处理，再调用B一次，查询一下之前的那次调用是否成功。例如A调用B更新订单状态，这时候成功了，延迟几秒后，A查询B，确认一下状态是否是自己刚刚期望的。上图中的D流程。

web service虽然主张交互之间采无状态方式，但是它并不是禁止采用有状态的交互。WEB SERVICE还是一种web技术，而web技术中的状态保存可能是最早被解决的问题之一了。在所有的web开发技术平台中，都有session机制，无论这些Session是通过IP，cookies, hidden input来实现，还是url sessionid来实现的，反正都有办法实现，请参阅所用平台的session支持机制就可以了。退一万步，你也可以通过在服务器中维护一个应用程序级的事务池来实现，未最后提交的事务对象都放在里面，每一个事务对象都给定一个唯一个的标志ID来识别，形成一个字典对象池。如果启动事务成功，则把此事务的ID返回给调用者执有，做第二段提交时，把事务的ID做为参数提交就是了。（随便提一下，用这种方法时，千万不能把对象的指针、句柄、引用什么的平台相关的值交给客户方，倒不是害怕安全问题，而是这些值在分布系统中是没有意义的，上次返回的指针没准早被垃圾收集机挪到其他地方去了）

    无论如何，webservice在通信层上是一种无连接的协议，每两次调用之间，tcp连接是断开的，因此，一但采用session机制来管理上下文，你就必须为这些session的生命期负责。试想，如果一个事务上下文已经开启，而此时客户方系统却突然当机了，这时会出什么事情？在同一个应用程序域中，客户方的当机会让连接中断，服务器立即就会中断并回退事务，但是在webservice里，状态管理机无法立即感觉到此事务的调用方已经失去控制，只能在一定的时间之后，才发现：“噫？这个事务已经N长时间没有人访问了！快快回退！”在ASP.net里，默认的状态超时时长大概是20分钟，JSP也差不多，阻塞了20分钟的事务对数据源是什么影响可想而止！因此，必须考虑合适的状态时长与事务隔离级别，以减小对数据源的性能影响。

# 微服务设计





# WebService/REST/Httpclient/HttpConcetions/RMI

1. RPC：RPC本身没有规范,但基本的工作机制是一样的，即：serialization/deserialization+stub+skeleton  
宽泛的讲，只要能实现远程调用，都是RPC，如:rmi .net-remoting ws/soap/rest hessian xmlrpc thrift potocolbuffer  
  
2. RMI是一种PRC.java的RMI就是java平台上的RPC技术方案。  
  
3. JMS是java平台上的消息规范。一般jms消息不是一个xml，而是一个java对象，很明显，jms没考虑异构系统，说白 了，JMS就没考虑非java的东西。但是好在现在大多数的jms provider（就是JMS的各种实现产品）都解决了异构问题。  
  
4. soap专注于远程服务调用，jms专注于信息交换。  
  
5. 大多数情况下soap是两系统间的直接交互（Consumer <--> Producer），而大多数情况下jms是三方系统交互（Consumer <- Broker -> Producer）。当然，JMS也可以实现request-response模式的通信，只要Consumer或Producer其中一方兼任 broker即可。  
  
6. 多数情况下，ws是同步的，jms是异步。虽然，ws也可以是异步的，而jms也可以是同步的。

# Nosql,Mongodb,Hbase,Cassandra

Hadoop获得了许多大数据应用的信誉，但实际情况是，NoSQL数据库是一直处于更加广泛部署和更广泛的发展中的技术。尽管选择Hadoop作为应用存储，相对来说更直接简单。但是，具体采用什么样的NoSQL数据库是个值得思考的问题，毕竟，还有超过100种的NoSQL数据库。

我们应该选择哪一种？

### ****选择倾向****

“任何像样规模的企业都会使用各种不同类型的数据存储技术，为应对各种不同类型的数据。”Martin Fowler认为，现实的情况是你没有足够的精力去学习更多的存储技术。

幸运的是，选择越来越容易，因为市场主要围绕在三个NoSQL数据库上：MongoDB，Cassandra（主要由DataStax开发的，诞生于Facebook），和HBase的（和Hadoop紧密关联在一起，也被相同社区开发出来）。

**补充一点**，我故意排除Redis。相对于大数据存储，它主要用于高速内存缓存数据应用。

从LinkedIn的451研究数据显示，市场上最具引力的是MongoDB、Cassandra和HBase：

这是LinkedIn的个人资料数据。我们认为是数据存储引擎，它通过收集工作、搜索等数据，来了解数据库的热门程度。而Oracle，SQL Server和MySQL的占据了统治地位，MongoDB的（第5位），Cassandra（第9位），和HBase的（第15位）。

为了更好解释为什么这三个数据库技术的如此耀眼，我问的每一个具有代表性的人，以确定它们成功关键因素：Kelly Stirman，MongoDB的产品总监;Patrick McFadin，DataStax的Cassandra首席布道师;和Justin Kestelyn，Cloudera高级总监。

但首先，我们需要了解为什么使用NoSQL的原因。

### ****世界由非结构化数据构成****

我们生活在一个数据越来越丰富的世界里，但是这些数据都不能整齐的展示在一个RDBMS（Relational Database Management System，关系数据库管理系）的行和列中。移动、社交和云计算催生了庞大的海量的数据。根据估计，世界上90％的数据是在过去两年中被创造，以及80％的商业数据是非结构化的。更重要的是，非结构化数据的增长速度是结构化数据的两倍。

随着世界的变化，数据管理要求开始超越传统的关系型数据库的有效范围。最早关注这个问题解决方案的机构，包括Web技术的先驱、政府机构、从事信息技术服务的公司。

现在越来越多，形形色色的公司都希望利用类似的NoSQL和Hadoop作为替代品：通过NoSQL来建立业务运营应用，以及Hadoop来创建数据挖掘的应用程序，来帮助公司对商业数据提供有力的研究。

### ****MongoDB：源于开发人员，为开发人员服务****

在众多NoSQL的方案中，MongoDB的Stirman指出，MongoDB的瞄准了适合各种应用的平衡的方法。它的功能接近于传统的关系型数据库，MongoDB的用户不仅可以利用其横向扩展机器的云基础架构的优势，并且，因为它能够轻松定义各种灵活的数据模型，所以可以支持不同类型的数据集存储。

MongoDB通常是开发人员第一个尝试的NoSQL数据库，因为它是很容易学习。Will Shulman，MongoLab（一个MongoDB服务提供商）的CEO，是这样说的：

MongoDB中的成功在很大程度上是因为它数据结构存储的创新，让我们更容易和更具表现力地定义我们应用程序中的数据模型。在通常开发和应用场景中，和原有数据库具有相同的基本数据模型是有极大好处的，因为它简化了应用程序开发的任务，另一方面，消除了复杂的数据格式代码转换层。

当然，像任何其他技术一样，MongoDB中都有其长处和短处。 MongoDB是专门为OLTP（On-Line Transaction Processing，联机事务处理系统）模式。如果您需要复杂的事务处理，它不是一个好的选择。然而，MongoDB的简单性使其成为一个优秀的存储。

（注：MongoDB以文档的形式存储数据，不支持事务和表连接。因此查询的编写、理解和优化都容易得多。）

### ****Cassandra：规模化安全运行****

三种数据库中，至少两种数据库具有简单特性：开发简单，操作简便。而MongoDB赢得人心的原因是简单的开发应用，Cassandra赢得人心是因为易于管理的规模。

DataStax的McFadin告诉我，用户往往倾向于使用Cassandra ，是因为特别在大规模集群下，增强一个关系型数据的性能、可靠性是非常困难的。一位前甲骨文DBA，McFadin是兴高采烈地发现，“复制和可扩放性是基础”，Cassandra 特点是从一开始设计就解决这个问题。

在RDBMS中的世界，数据库功能，拓展和复制对很多开发者用户来说，是一个难题。这个问题在过往的企业规模小的时候，不是一个大问题。而在今天，它很迅速地成为大问题。

我从McFadin和其他人那里获知，Cassandra在机器拓展部署上，表现特别出色。Cassandra自带的备份机制，保证各个数据中心的数据安全。至于增加容量到集群，“你只需启动一台新机器，并告诉Cassandra那里的新节点，”McFadin说，“然后，它完成其他剩下的事情。”

优秀的可拓展性，加上出色的写入和可观的查询性能，加起来成为Cassandra高性能的核心。

NoSQL的一篇文章认为Cassandra在集群规模管理方面非常出色，但它需要一个博士学位才能上手。事实并非如此，McFadin坚持认为：  
在复制、读取和写入是故意简单。你可以在几个小时内学会Cassandra的核心功能。在部署这项新技术的时候，为给开发者带来很多的信心，因为比较少引入“黑盒子”内的技术细节和复杂的故障模式原理。

这意味着主要的开发成本，是对Cassandra数据模型的理解，以及如何结合您的应用程序。鉴于Cassandra的CQL查询语言（类似于SQL，实际上不是SQL），McFadin说，学习这个也不困难。

更重要的是，他告诉我，“Cassandra回报给你的是，在一个数据库中：没有戏剧性的场景（故障）出现。这就是用户喜欢使用Cassandra的原因。”

### ****HBase：Hadoop的知心伙伴****

HBase，像Cassandra一样是个通过key-value面向列存储的服务。因为它和Hadoop有着“共同血统”，被广泛使用。事实上，正如Cloudera的Kestelyn所说的那样，“HBase提供了一个基于记录的存储层，能够快速随机读取和写入数据，正好弥补了Hadoop的缺陷，Hadoop侧重系统吞吐量，而牺牲I / O读取效率为代价。”

Kestelyn接着说：

更改有效录入到内存中，以达到最大的访问量，同时将数据保存到HDFS。这种设计使基于Hadoop的EDH（enterprise data hub，企业数据中心）服务，能够实时完成随机读写存储数据，但仍拥有HDFS的高容错性和耐用性。

Hadoop的亲和力，不是HBase数据库中的人气排名不断上升的唯一原因。类似Cassandra，HBase是Google的Bigtable的开源实现转化成的数据库，天然被设计为高可扩展性。

Hbase可以利用任何数量服务器的磁盘、内存和CPU资源，同时拥有极佳的扩展功能，如自动分片。当系统负载和性能要求不断增加，HBase的可通过简单增加服务器节点的方式无限拓展。 HBase从底层设计上保证，在确保数据一致性的同时，提供最佳性能。

但规模不是它的唯一用途。Kestelyn指出，“由于它与Hadoop的生态系统紧密集成，对于用户和应用程序来说，数据是容易获取的，可以通过SQL的方式查询（使用Cloudera的Impala，Phoenix，或Hive），甚至自由文本搜索（使用Cloudera Search）。“因此，HBase为开发人员提供了一种方法，利用现有通用的SQL语言，来建立在一个更成熟的分布式数据库。

每种数据库技术都有自己的长处和不足，但这里评论的三种数据库，在大数据技术领域，占据了重要的位置。虽然未来可能还有一种全新的NoSQL数据库技术会挑战它们前三的位置，但目前的现实是，许多开发人员以及一批强大的成熟企业已经做出了它们的选择：MongoDB、Cassandra 和 HBase。

# 正向代理&反向代理&Nginx&VPN&dns

正向代理,也就是传说中的代理,他的工作原理就像一个跳板。简单的说,我是一个用户,我访问不了某网站,

但是我能访问一个代理服务器,这个代理服务器呢,他能访问那个我不能访问的网站,于是我先连上代理服务

器,告诉他我需要那个无法访问网站的内容,代理服务器去取回来,然后返回给我。从网站的角度,只在代理

服务器来取内容的时候有一次记录。结论就是，正向代理，是一个位于客户端和原始服务器(origin

server)之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，

然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端必须要进行一些特别的设置才能使用

正向代理。

例用户访问 http://ooxx/readme但ooxx上并不存在readme页面他是偷偷从另外一台服务器上取回来,然后

作为自己的内容吐给用户。但用户并不知情这很正常,用户一般都很笨这里所提到的 ooxx这个域名对应的

服务器就设置了反向代理功能结论就是 反向代理正好相反，对于客户端而言它就像是原始服务器，并且客

户端不需要进行任何特别的设置。客户端向反向代理 的命名空间(name-space)中的内容发送普通请求，接

着反向代理将判断向何处(原始服务器)转交请求，并将获得的内容返回给客户端，就像这些内容 原本就是

它自己的一样。

正向代理的典型用途是为在防火墙内的局域网客户端提供访问Internet的途径。正向代理还可以使用缓冲

特性减少网络使用率。反向代理的典型用途是将 防火墙后面的服务器提供给Internet用户访问。反向代理

还可以为后端的多台服务器提供负载平衡，或为后端较慢的服务器提供缓冲服务。

1、nginx 和 apache

@@nginx相对于apache的优点：

（1）轻量级，同样起web 服务，比apache 占用更少的内存及资源

（2）抗并发，nginx 处理请求是异步非阻塞的，而apache 则是阻塞型的，在高并发下nginx 能保持低资源低消耗高性能

（3）高度模块化的设计，编写模块相对简单

（4）社区活跃，各种高性能模块出品迅速啊

@@apache 相对于nginx 的优点：

（1）rewrite ，比nginx 的rewrite 强大

（2）模块超多，基本想到的都可以找到

（3）少bug ，nginx 的bug 相对较多 ，稳定

存在就是理由，一般来说，需要性能的web 服务，用nginx 。

如果不需要性能只求稳定，那就apache 吧。后者的各种功能模块实现得比前者，例如ssl 的模块就比前者好，可配置项多。

2、Nginx的三种场景

（1）作为 Web 服务器：相比 Apache，Nginx 使用更少的资源，支持更多的并发连接，体现更高的效率，这点使 Nginx 尤其受到虚拟主机提供商的欢迎。在高连

接并发的情况下，Nginx是Apache服务器不错的替代品: Nginx在美国是做虚拟主机生意的老板们经常选择的软件平台之一. 能够支持高达 50,000 个并发连接数

的响应, 感谢Nginx为我们选择了 epoll and kqueue 作为开发模型.

（2）Nginx作为负载均衡服务器: Nginx 既可以在内部直接支持 Rails 和 PHP 程序对外进行服务, 也可以支持作为 HTTP代理 服务器对外进行服务. Nginx采用C进行

编写, 不论是系统资源开销还是CPU使用效率都比 Perlbal 要好很多.

（3）作为邮件代理服务器: Nginx 同时也是一个非常优秀的邮件代理服务器（最早开发这个产品的目的之一也是作为邮件代理服务器）, Last.fm 描述了成功并且美妙的使用经验.

Nginx 是一个安装非常的简单 , 配置文件非常简洁（还能够支持perl语法）, Bugs 非常少的服务器: Nginx 启动特别容易, 并且几乎可以做到7\*24不间断运行，即使运行数个月也

不需要重新启动. 你还能够不间断服务的情况下进行软件版本的升级 .

3、Nginx 配置简洁, Apache 复杂 ；Nginx 静态处理性能比 Apache 高 3倍以上 ；Apache 对 PHP 支持比较简单，Nginx 需要配合其他后端用

Apache 的组件比 Nginx 多 ；现在 Nginx 才是 Web 服务器的首选

4、最核心的区别在于apache是同步多进程模型，一个连接对应一个进程；nginx是异步的，多个连接（万级别）可以对应一个进程

5、nginx处理静态文件好,耗费内存少.但无疑apache仍然是目前的主流,有很多丰富的特性.所以还需要搭配着来.当然如果能确定nginx就适合需求,那么使用nginx会是更经济的方式.

6、从个人过往的使用情况来看，nginx的负载能力比apache高很多。最新的服务器也改用nginx了。而且nginx改完配置能-t测试一下配置有没有问题，apache重启的时候发现配置出

错了，会很崩溃，改的时候都会非常小心翼翼现在看有好多集群站，前端nginx抗并发，后端apache集群，配合的也不错。

7、nginx处理动态请求是鸡肋，一般动态请求要apache去做，nginx只适合静态和反向。

8、從我個人的經驗來看，nginx是很不錯的前端服務器，負載性能很好，在老奔上開nginx，用webbench模擬10000個靜態文件請求毫不吃力。apache對php等語言的支持很好

，此外apache有強大的支持網路，發展時間相對nginx更久，bug少但是apache有先天不支持多核心處理負載雞肋的缺點，建議使用nginx做前端，後端用apache。大型網站建

議用nginx自代的集群功能

9、Nginx优于apache的主要两点：1.Nginx本身就是一个反向代理服务器 2.Nginx支持7层负载均衡；其他的当然，Nginx可能会比apache支持更高的并发，但是根据NetCraf

t的统计，2011年4月的统计数据，Apache依然占有62.71%，而Nginx是7.35%，因此总得来说，Aapche依然是大部分公司的首先，因为其成熟的技术和开发社区已经也是非常

不错的性能。

10、你对web server的需求决定你的选择。大部分情况下nginx都优于APACHE，比如说静态文件处理、PHP-CGI的支持、反向代理功能、前端Cache、维持连接等等。

在Apache+PHP（prefork）模式下，如果PHP处理慢或者前端压力很大的情况下，很容易出现Apache进程数飙升，从而拒绝服务的现象。

11、可以看一下nginx lua模块：https://github.com/chaoslaw...apache比nginx多的模块，可直接用lua实现apache是最流行的，why？大多数人懒得更新到nginx或者学新事物

12、对于nginx，我喜欢它配置文件写的很简洁，正则配置让很多事情变得简单运行效率高，占用资源少，代理功能强大，很适合做前端响应服务器

13、Apache在处理动态有优势，Nginx并发性比较好，CPU内存占用低，如果rewrite频繁，那还是Apache吧

十、使用场景

1、http服务器。Nginx是一个http服务可以独立提供http服务。可以做网页静态服务器。

2、虚拟主机。可以实现在一台服务器虚拟出多个网站，例如个人网站使用的虚拟机。

3、反向代理，负载均衡。当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使用nginx做反向代理。

并且堕胎服务器可以平均分担负载，不会应为某台服务器负载高宕机而某台服务器闲置的情况。

一、基础概述

0 Nginx是一款轻量级的Web 服务器/反向代理服务器及电子邮件(IMAP/POP3)代理服务器/负载均衡服务器，并在一个BSD-like 协议下发行。由俄罗斯的程序设计师Igor Sysoev所开发，供俄国大型的

入口网站及搜索引擎Rambler(俄文：Рамблер)使用。其特点是占有内存少，并发能力强，事实上nginx的并发能力确实在同类型的网页服务器中表现较好.

中国大陆使用nginx网站用户有：新浪、网易、 腾讯等。

1 正向代理：是一个位于客户端和原始服务器(origin server)之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内

容返回给客户端。客户端才能使用正向代理，而且必须要进行一些特别的设置才能使用正向代理。。正向代理是一种最终用户知道并主动使用的代理方式。 正向代理的典型用途是为在防火墙内的局域网客户端

提供访问Internet的途径。正向代理还可以使用缓冲特性减少网络使用率。正向代理允许客户端通过它访问任意网站并且隐藏客户端自身，因此你必须采取安全措施以确保仅为经过授权的客户端提供服务。

2 反向代理：是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个反向

代理服务器。反向代理对用户来说是透明的，用户是感知不到的。 反向代理的典型用途是将防火墙后面的服务器提供给Internet用户访问。反向代理还可以为后端的多台服务器提供负载平衡，或为后端较慢的服

务器提供缓冲服务。

3 Nginx是目前比较主流的HTTP反向代理服务器（其企业版提供了基于TCP层的反向代理插件），对于构建大型分布式web应用，具有举足轻重的作用。简单来说，nginx有2个主要的功能：动/静态资源分离、负载均衡

4 动/静态资源分离：nginx支持正则表达式以区分静态资源或者动态资源，其中动态资源可以进一步转发给后端的proxy server，而静态资源则可以在nginx层面使用本地缓存策略或者重定向（类CDN）到其他nginx上。

负载均衡：对于动态资源而言，如果有多个proxy server，那么nginx将会根据一定的算法选择合适的server，并转发请求，最终将客户端request相对均衡的分发给多个server。

5 Nginx作为"单点"，面向客户端请求，并将请求转发给后端的某个server，它应该被架设在物理资源较为优越的机器上，比如8U物理机，32核心，64G内存，对磁盘要求相对较低，对CPU、内存、网卡带宽有较高的要求，

因为nginx不仅需要和客户端请求建立链接，而且还需要与后端proxy server建立链接并且负责流量输入、输出，这种双倍的链接建立，就要求机器具有较高的内存和CPU，最终为所有proxy server流量的总和，那么它应该具有更高的网卡带宽。

为了避免资源竞争，应该避免nginx和web server部署在同一个节点上，因为web server通常为CPU和内存高耗型，这会大大降低nginx的代理能力。

6 1) 在中小型应用中（PV在KW级别，单一垂直web应用），通常一个nginx代理多个（组）server即可。

2）对于大中型应用，一个nginx将无法支撑全部的流量，我们将会采用多个nginx代理（复制了1）中的架构模型），并在nginx前端继续构建高性能的分流设备，比如LVS、Haproxy等更低层的软/硬件负载均衡器，这种负

载均衡器通常只是"转发"，而不涉及到流量的输出，所以转发效率将会更高，承载能力更强。

3）无论何时，我们也不希望nginx存在单点故障问题，那么通常我们还需要使用keepalived（其他同类型技术，VIP）来提高nginx节点的可用性，即Master-backup模式。

4）当有多个nginx时，为了提升后端server的代理能力，通常还会让多个nginx之间交叉重叠代理后端的server。

Request

|

dns

|

| |

nginx nginx

(tomcat,tomcat) (tomcat,tomcat)

# 微服务&Zookeeper&Spring boot&doubbo

  Spring cloud适应于云端服务，也适用于企业信息化SOA建设。spring boot也是restful微服务开发的利器。但对于内网服务，即服务与服务之间的调用，spring并没有去刻意封装，也许他们认为已经没有必要了，因为已经有了thrift、ice等强大的框架。

    如果是用spring boot本身提供的restful服务作为服务与服务之间的调用，效率低很多，thrift的效率大概是restful的100-1000倍左右。本篇既是基于spring boot框架，结合thrift和zookeeper实现的一个简单微服务框架，服务与服务之间使用thrift通信（thrift既是通信方式也是数据压缩方式）。

    本demo一共包括三个工程：

    cloud-thrift-server：服务提供方

    cloud-thrift-interface：接口及传输对象定义

    cloud-thrift-client：服务调用方

    开源代码地址：<http://git.oschina.net/zhou666/spring-cloud-7simple>

1）建立thrift接口定义文档

    namespace java cloud.simple.service

    struct UserDto {

      1: i32 id

      2: string username

    }

     service UserService {

      UserDto getUser()

    }

    接口定义完后，使用thrift命令生成对应的java文件，主要生成两个文件，分别是UserService.java和UserDto.java，把这两个文件放入cloud-thrift-interface工程，因为客户端也需要这个接口定义。

2）实现thrift服务注册

    在服务的提供端需要实现接口，并且还要把实现类注册到thrift服务器。

    UserService.Processor processor = new UserService.Processor(

                       new UserServiceImpl());

         TServer server = new TThreadPoolServer(new TThreadPoolServer.Args(

                       tServerTransport()).processor(processor));

    UserServiceImpl就是接口实现类，将其注册金Tserver。

    注册完服务后，需要启动Tserver，很显然这个需要在线程里启动。

    executor.execute(new Runnable() {

                @Override

                public void run() {

                       tServer().serve();

                }

    });

3） 使用zookeeper进行服务名称注册

    上面是注册具体的服务执行类，这一步是将服务的实例注册进zookeeper，这样才能实现负载均衡。让客户端可以根据服务实例列表选择服务来执行。当然这里只需要注册服务所在服务器的IP即可，因为客户端只要知道IP，也就知道访问那个IP下的该服务。

    String servicePath = "/"+serviceName ;// 根节点路径

       ZkClient zkClient = new ZkClient(serverList);

       boolean rootExists = zkClient.exists(servicePath);

       if (!rootExists) {

           zkClient.createPersistent(servicePath);

       }

       InetAddress addr = null;

       try {

           addr = InetAddress.getLocalHost();

       } catch (UnknownHostException e) {

           e.printStackTrace();

       }

       String ip = addr.getHostAddress().toString();

       String serviceInstance = System.nanoTime() +"-"+ ip;

       // 注册当前服务

       zkClient.createEphemeral(servicePath + "/" + serviceInstance);

       System.out.println("提供的服务为：" + servicePath + "/" + serviceInstance);

    要注意这里使用zkClient.createEphemeral建立临时节点，如果这台服务器宕机，这个临时节点是会被清除的，这样客户端在访问时就不会再选择该服务器上的服务。

4） 客户端更新服务列表

    客户端需要能及时的监听服务列表的变化并作出负载均衡，我们用如下方式监听服务列表的变化：

    // 注册事件监听

         zkClient.subscribeChildChanges(servicePath, new IZkChildListener() {

                // @Override

                public void handleChildChange(String parentPath,

                              List<String> currentChilds) throws Exception {

                       // 实例(path)列表:当某个服务实例宕机，实例列表内会减去该实例

                       for (String instanceName : currentChilds) {

                              // 没有该服务，建立该服务

                              if (!serviceMap.containsKey(instanceName)) {

                                     serviceMap.put(instanceName,createUserService(instanceName));

                              }

                       }

                       for (Map.Entry<String, UserService.Client> entry : serviceMap.entrySet()) {

                              // 该服务已被移除

                              if (!currentChilds.contains(entry.getKey())) {

                                     serviceMap.remove(entry.getKey());

                              }

                       }

                       System.out.println(parentPath + "事件触发");

                }

    });

    有了服务列表，客户端在调用服务的时候就可以采用负载均衡的方式了，在这里使用最简单的随机方式：

         public UserService.Client getBalanceUserService(){

         Map<String, UserService.Client> serviceMap =ZooKeeperConfig.serviceMap;

         //以负载均衡的方式获取服务实例

         for (Map.Entry<String, UserService.Client> entry : serviceMap.entrySet()) {

                System.out.println("可供选择服务:"+entry.getKey());

         }

         int rand=new Random().nextInt(serviceMap.size());

         String[] mkeys = serviceMap.keySet().toArray(new String[serviceMap.size()]);

         return serviceMap.get(mkeys[rand]);

  }

   从SpringBoot 项目名称中的Boot可以看出来，SpringBoot 的作用在于创建和启动新的基于Spring框架的项目。它的目的是帮助开发人员很容易的创建出独立运行和产品级别的基于Spring框架的应用。SpringBoot 会选择最适合的Spring子项目和第三方开源库进行整合。大部分SpringBoot 应用只需要非常少的配置就可以快速运行起来。  
    Spring Boot 包含的特性如下：  
创建可以独立运行的Spring应用。  
    直接嵌入Tomcat 或Jetty服务器，不需要部署WAR 文件。  
    提供推荐的基础POM文件来简化ApacheMaven 配置。  
    尽可能的根据项目依赖来自动配置Spring框架。  
    提供可以直接在生产环境中使用的功能，如性能指标、应用信息和应用健康检查。  
    没有代码生成，也没有XML 配置文件  
    回首我们以前使用Spring框架的时候，我们需要首先在（如果你使用Maven的话）pom文件中增加对相关的的依赖（使用gradle来构建的话基本也一样）然后新建Spring相关的xml文件，而且往往那些xml文件还不会少。然后继续使用tomcat或者jetty作为容器来运行这个工程。基本上每次创建一个新的项目都是这么一个流程，而我们有时候仅仅想快速的创建一个Springweb工程来测试一些东西，或者是希望能节省时间。  
    Spring boot让所有Spring开发变得更快，且让更多的人更快的进行Spring入门体验，提供“starter”POM来简化我们的Maven配置（也就是说使用SpringBoot只有配合maven/gradle等这种依赖管理工具才能发挥它的能力），不像以前，构建一个springmvc项目需要进行好多配置等  
开箱即用，快速开始需求开发而不被其他方面影响（如果可能会自动配置Spring）  
    提供一些非功能性的常见的大型项目类特性（如内嵌服务器、安全、度量、健康检查、外部化配置），如可以直接地内嵌Tomcat/Jetty（不需要单独去部署war包）绝无代码生成，且无需XML配置

  Spring boot 是一个spring 框架，主要的作用是简化使用spring开发的难度，减少配置文件，提高开发效率。使用spring boot开发之后，最直观的感受是配置文件变少了，原来使用spring框架可能需要的多个配置文件都被简化了，现在一个配置文件就可以全部搞定。而且这个配置文件里配置也很简单，都是key=value的形式，没有原来配置文件里复杂的属性配置，没有了繁琐复杂，降低了开发难度。

   使用spring boot 开发推荐使用maven 之类的包依赖管理技术，一个pom文件就可以导入需要的所有jar包，后面添加新的jar也很方便。想要学习一个东西，最有效的方法就是去使用它。学习spring bootk可以通过动手写一个小demo来快速入门。首先新建一个mave项目，然后在pom文件中加入以下内容：

**Java代码**[**IMG_256**](http://build.iteye.com/blog/javascript:void())

1. <parent>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-parent</artifactId>
4. <version>1.3.1.RELEASE</version>
5. </parent>
7. <dependencies>
8. <dependency>
9. <groupId>org.springframework.boot</groupId>
10. <artifactId>spring-boot-starter-web</artifactId>
11. </dependency>
12. </dependencies>

 保存之后,maven会去自动下载spring boot需要的jar包

新建一个StartController类，内容如下：

**Java代码**[**IMG_257**](http://build.iteye.com/blog/javascript:void())

1. **package** com.springboot.summary.start;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4. **import** org.springframework.stereotype.Controller;
5. **import** org.springframework.web.bind.annotation.RequestMapping;
6. **import** org.springframework.web.bind.annotation.ResponseBody;
8. @Controller
9. @EnableAutoConfiguration
10. **public** **class** StartController
11. {
13. @RequestMapping("/start")
14. @ResponseBody
15. String start() {
16. **return** "Hello World!";
17. }
18. **public** **static** **void** main( String[] args )
19. {
20. SpringApplication.run(StartController.**class**, args);
21. }
22. }

     可以看到，StartController类中包含了一个main函数，启动这个main函数，就会启动spring boot内置tomcat.在本机浏览器输入 <http://localhost:8080/start> 后，就可以看见hello world!的输出。最后附上项目源码，以供参考。

# NIO框架

## 了解Java NIO

传统的并发型服务器设计是利用阻塞型网络I/O 以多线程的模式来实现的，然而由于系统常常在进行网络读写时处于阻塞状态，会大大影响系统的性能；自Java1. 4 开始引入了NIO(新I/O) API，通过使用非阻塞型I/O，实现流畅的网络读写操作，为开发高性能并发型服务器程序提供了一个很好的解决方案。这就是java nio。

## 传统阻塞型网络 I/O的不足

Java 平台传统的I/O 系统都是基于Byte（字节）和Stream（数据流）的，相应的I/O操作都是阻塞型的，所以服务器程序也采用阻塞型I/O 进行数据的读、写操作。传统的设计为了实现服务器程序的并发性要求，系统由一个单独的主线程来监听用户发起的连接请求，一直处于阻塞状态；当有用户连接请求到来时，程序都会启一个新的线程来统一处理用户数据的读、写操作。在阻塞的网络编程方式中，针对于每一个单独的网络连接，都必须有一个线程对应的绑定该网络连接，进行网络字节流的处理。看看以前传统写法：  
  try {  
   ServerSocket ssc = new ServerSocket(8086);  
   while (true) {  
    //阻塞  
    Socket s = ssc.accept();  
    try {  
     //每来一个客户端就开启一个读线程  
     new ReadThread(s).start();  
     //同时每来一个客户端就开启一个写线程  
     new WriteThread(s).start();  
    } catch (Exception e) {  
     e.printStackTrace();  
    }  
   }  
  } catch (IOException e) {  
   e.printStackTrace();  
  }

以上就是传统阻塞模式写法，在这段代码中，有三个阻塞的方法，是ServerSocket的accept()方法，InputStream的read()方式以及OutputStream的write()方式。因此我们需要三个的线程（主线程也是一个）分别进行处理，要每一个客户端分配二个线程来处理输入、输出数据。这样如果有大量的连接存在，就存在大量的线程，其线程与客户机的比例近似为1:1，随着线程数量的不断增加，服务器启动了大量的并发线程，会大大加大系统对线程的管理开销，这将成为吞吐量瓶颈的主要原因，而大量的线程又都阻塞在read()或者write()方法，同时CPU又需要来回频繁的在这些线程中间调度和切换，必然带来大量的系统调用和资源竞争。但传统模式的也是有它的优点的，优点就是简单、实用、易管理。

对于并发型服务器，系统用在阻塞型I/O 等待和线程间切换的时间远远多于CPU 在内存中处理数据的时间，因此传统的阻塞型I/O 已经成为制约系统性能的瓶颈。Java1.4 版本后推出的NIO 工具包，提供了非阻塞型I/O 的异步输入输出机制，为提高系统的性能提供了可实现的基础机制。

## NIO工作原理

通过上面的讨论，我们可以看出在并发型服务器程序中使用NIO，实际上是通过网络事件驱动模型实现的。我们应用Select机制，不用为每一个客户端连接新启线程处理，而是将客户端Socket连接注册到特定的Selector对象上，这就可以在单线程中利用Selector对象管理大量并发的网络连接，更好的利用了系统资源；采用非阻塞I/O的通信方式，不要求阻塞等待I/O 操作完成即可返回，从而减少了管理I/O 连接导致的系统开销，大幅度提高了系统性能。

当有读或写等任何注册的事件发生时，可以从Selector中获得相应SelectionKey，从SelectionKey 中可以找到发生的事件和该事件所发生的具体的SelectableChannel，以获得客户端发送过来的数据。由于在非阻塞网络I/O 中采用了事件触发机制，处理程序可以得到系统的主动通知，从而可以实现底层网络I/O无阻塞、流畅地读写，而不像在原来的阻塞模式下处理程序需要不断循环等待。使用NIO，可以编写出性能更好、更易扩展的并发型服务器程序。

应用 NIO 工具包，基于非阻塞网络I/O设计的并发型服务器程序与以往基于阻塞I/O的实现程序有很大不同，在使用非阻塞网络I/O的情况下，程序读取数据和写入数据的时机不是由程序员控制的，而是Selector 决定的。

通过使用NIO 工具包进行并发型服务器程序设计，一个或者很少几个Socket 线程就可以处理成千上万个活动的Socket 连接，大大降低了服务器端程序的开销；同时网络I/O 采取非阻塞模式，线程不再在读或写时阻塞，操作系统可以更流畅的读写数据并可以更有效地向CPU 传递数据进行处理，以便更有效地提高系统的性能。

## NIO框架

**Mina：**  
  
Mina(Multipurpose Infrastructure for Network Applications) 是 Apache 组织一个较新的项目，它为开发高性能和高可用性的网络应用程序提供了非常便利的框架。当前发行的 Mina 版本2.04支持基于 Java NIO 技术的 TCP/UDP 应用程序开发、串口通讯程序，Mina 所支持的功能也在进一步的扩展中。目前，正在使用 Mina的应用包括：Apache Directory Project、AsyncWeb、AMQP（Advanced Message Queuing Protocol）、RED5 Server（Macromedia  Flash Media RTMP）、ObjectRADIUS、 Openfire等等。  
  
**Netty：**  
  
Netty是一款异步的事件驱动的网络应用框架和工具，用于快速开发可维护的高性能、高扩展性协议服务器和客户端。也就是说，Netty是一个NIO客户端/服务器框架，支持快速、简单地开发网络应用，如协议服务器和客户端。它极大简化了网络编程，如TCP和UDP套接字服务器。

**Grizzly：**  
  
 Grizzly是一种应用程序框架，专门解决编写成千上万用户访问服务器时候产生的各种问题。使用JAVA NIO作为基础，并隐藏其编程的复杂性。容易使用的高性能的API。带来非阻塞socketd到协议处理层。利用高性能的缓冲和缓冲管理使用高性能的线程池。  
  
  
OK，我们现在可以看看三者的简单对比了。  
  
  
首先，从设计的理念上来看，Mina的设计理念是最为优雅的。当然，由于Netty的主导作者与Mina的主导作者是同一人，出自同一人之手的Netty在设计理念上与Mina基本上是一致的。而Grizzly在设计理念上就较差了点，几乎是Java NIO的简单封装。  
  
其次，从项目的出身来看，Mina出身于开源界的大牛Apache组织，Netty出身于商业开源大亨Jboss，而Grizzly则出身于土鳖Sun公司。从其出身可以看到其应用的广泛程序，到目前为止，我见到业界还是使用Mina多一些，而Netty也在慢慢的应用起来，而Grizzly则似乎只有Sun自已的项目使用了，如果还有其他的公司或开源项目在使用，那就算我孤陋寡闻。  
  
  
最后，从入门的文档来说，由于Mina见世时间相对较长，官方以及民间的文档与入门示例都相当的多。Netty的官方文档也做得很好，而民间文档就要相对于Mina少一些了。至于Grizzly，不管是官方还是民间，都很少见到其文

# 分布式，集群，微服务

集群是个物理形态，分布式是个工作方式。

分布式：一个业务分拆多个子业务，部署在不同的服务器上

集群：同一个业务，部署在多个服务器上

1：分布式是指将不同的业务分布在不同的地方。而集群指的是将几台服务器集中在一起，实现同一业务。

分布式中的每一个节点，都可以做集群。而集群并不一定就是分布式的。

举例：就比如新浪网，访问的人多了，他可以做一个群集，前面放一个响应服务器，后面几台服务器完成同一业务，如果有业务访问的时候，响应服务器看哪台服务器的负载不是很重，就将给哪一台去完成。

而分布式，从窄意上理解，也跟集群差不多，但是它的组织比较松散，不像集群，有一个组织性，一台服务器垮了，其它的服务器可以顶上来。

分布式的每一个节点，都完成不同的业务，一个节点垮了，那这个业务就不可访问了。

2：简单说，分布式是以缩短单个任务的执行时间来提升效率的，而集群则是通过提高单位时间内执行的任务数来提升效率。

例如：如果一个任务由 10 个子任务组成，每个子任务单独执行需 1 小时，则在一台服务器上执行该任务需 10 小时。

采用分布式方案，提供 10 台服务器，每台服务器只负责处理一个子任务，不考虑子任务间的依赖关系，执行完这个任务只需一个小时。(这种工作模式的一个典型代表就是 Hadoop 的 Map/Reduce 分布式计算模型）

而采用集群方案，同样提供 10 台服务器，每台服务器都能独立处理这个任务。假设有 10 个任务同时到达，10 个服务器将同时工作，1 小时后，10 个任务同时完成，这样，整身来看，还是 1 小时内完成一个任务！

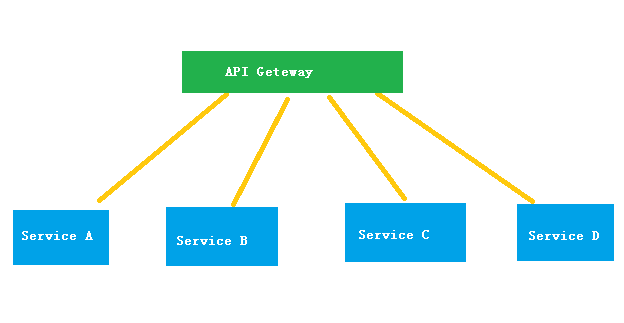
好的设计应该是分布式和集群的结合，先分布式再集群，具体实现就是业务拆分成很多子业务，然后针对每个子业务进行集群部署，这样每个子业务如果出了问题，整个系统完全不会受影响。

以上摘自：[分布式与集群的区别是什么？](https://www.zhihu.com/question/20004877) 分布式概念参考：[分布式系统的特点以及设计理念](http://www.infoq.com/cn/articles/features-and-design-concept-of-distributed-system)

另外，还有一个概念和分布式比较相似，那就是微服务。

**微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力。**

## 1.分布式



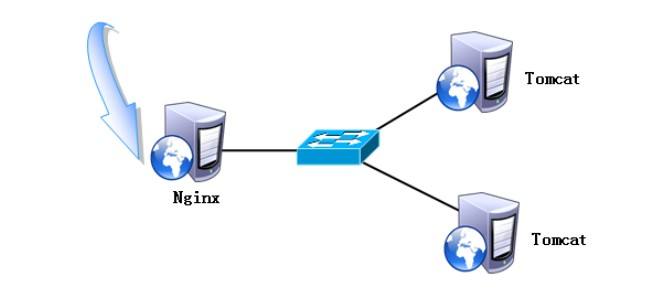
将一个大的系统划分为多个业务模块，业务模块分别部署到不同的机器上，各个业务模块之间通过接口进行数据交互。区别分布式的方式是根据不同机器不同业务。

**上面：service A、B、C、D 分别是业务组件，通过API Geteway进行业务访问。**

注：分布式需要做好事务管理。

分布式事务可参考：[微服务架构的分布式事务解决方案](https://my.oschina.net/838398/blog/761261)

## 2.集群模式



集群模式是不同服务器部署同一套服务对外访问，实现服务的负载均衡。区别集群的方式是根据部署多台服务器业务是否相同。

注：集群模式需要做好session共享，确保在不同服务器切换的过程中不会因为没有获取到session而中止退出服务。

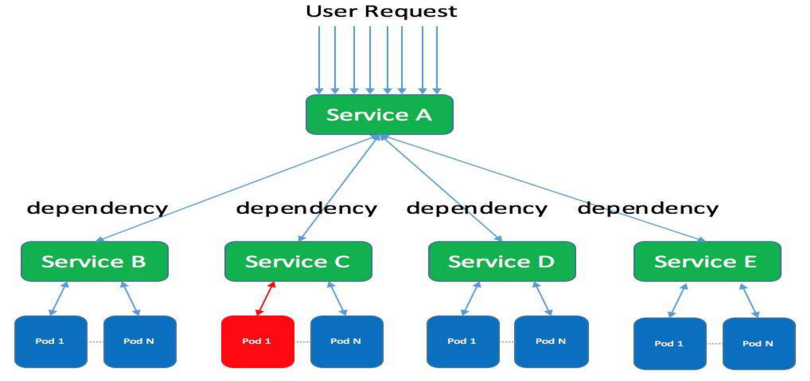
一般配置Nginx\*的负载容器实现：静态资源缓存、Session共享可以附带实现，Nginx支持5000个并发量。

## 3.分布式是否属于微服务？

答案是肯定的。[**微服务**](http://lib.csdn.net/base/microservice)的意思也就是将模块拆分成一个独立的服务单元通过接口来实现数据的交互。

## 4.微服务架构

微服务的设计是为了不因为某个模块的升级和BUG影响现有的系统业务。微服务与分布式的细微差别是，微服务的应用不一定是分散在多个服务器上，他也可以是同一个服务器。



分布式和微服的[**架构**](http://lib.csdn.net/base/architecture)很相似，只是部署的方式不一样而已。

# kafka内部原理，三种方式，大小。

1如何收集这些巨大的信息

2如何分析它

3如何及时做到如上两点

以上几个挑战形成了一个业务需求模型，即生产者生产（produce）各种信息，消费者消费（consume）（处理分析）这些信息，而在生产者与消费者之间，需要一个沟通两者的桥梁-消息系统。从一个微观层面来说，这种需求也可理解为不同的系统之间如何传递消息。  
kafka-即是解决上述这类问题的一个框架，它实现了生产者和消费者之间的无缝连接。

@高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒

@可扩展性：kafka集群支持热扩展

@持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失

@容错性：允许集群中节点失败（若副本数量为n,则允许n-1个节点失败）

@高并发：支持数千个客户端同时读写

**话题（Topic）：**是特定类型的消息流。消息是字节的有效负载（Payload），话题是消息的分类名或种子（Feed）名。

**生产者（Producer）：**是能够发布消息到话题的任何对象。

**服务代理（Broker）：**已发布的消息保存在一组服务器中，它们被称为代理（Broker）或Kafka集群。

**消费者（Consumer）：**可以订阅一个或多个话题，并从Broker拉数据，从而消费这些已发布的消息。

# classloader。

ClassLoader翻译过来就是类加载器，普通的[Java](http://lib.csdn.net/base/java)开发者其实用到的不多，但对于某些框架开发者来说却非常常见。理解ClassLoader的加载机制，也有利于我们编写出更高效的代码。ClassLoader的具体作用就是将class文件加载到jvm虚拟机中去，程序就可以正确运行了。但是，jvm启动的时候，并不会一次性加载所有的class文件，而是根据需要去动态加载。想想也是的，一次性加载那么多jar包那么多class，那内存不崩溃。本文的目的也是学习ClassLoader这种加载机制。

## JAVA\_HOME

指的是你JDK安装的位置，一般默认安装在C盘，如

C:\Program Files\Java\jdk1.8.0\_91

## PATH

将程序路径包含在PATH当中后，在命令行窗口就可以直接键入它的名字了，而不再需要键入它的全路径,比如上面代码中我用的到javac和java两个命令。   
一般的

PATH=%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;%PATH%;

也就是在原来的PATH路径上添加JDK目录下的bin目录和jre目录的bin.

## CLASSPATH

CLASSPATH=.;%JAVA\_HOME%\lib;%JAVA\_HOME%\lib\tools.jar

* 1
* 1

一看就是指向jar包路径。   
需要注意的是前面的.;，.代表当前目录。

## 环境变量的设置与查看

设置可以右击我的电脑，然后点击属性，再点击高级，然后点击环境变量，具体不明白的自行查阅文档。

查看的话可以打开命令行窗口

echo %JAVA\_HOME%

echo %PATH%

echo %CLASSPATH%

Java语言系统自带有三个类加载器:   
**- Bootstrap ClassLoader**最顶层的加载类，主要加载核心类库，%JRE\_HOME%\lib下的rt.jar、resources.jar、charsets.jar和class等。另外需要注意的是可以通过启动jvm时指定-Xbootclasspath和路径来改变Bootstrap ClassLoader的加载目录。比如java -Xbootclasspath/a:path被指定的文件追加到默认的bootstrap路径中。我们可以打开我的电脑，在上面的目录下查看，看看这些jar包是不是存在于这个目录。

System.out.println(System.getProperty("sun.boot.class.path"));  
**- Extention ClassLoader**扩展的类加载器，加载目录%JRE\_HOME%\lib\ext目录下的jar包和class文件。还可以加载-D java.ext.dirs选项指定的目录。

System.out.println(System.getProperty("Java.ext.dirs"));  
**- Appclass Loader**也称为SystemAppClass 加载当前应用的classpath的所有类。

System.out.println(System.getProperty("java.class.path"));

**- 自定义Loader**也称为SystemAppClass 加载当前应用的classpath的所有类。

编写一个类继承自ClassLoader抽象类。

复写它的findClass()方法。

在findClass()方法中调用defineClass()。

1. 编写一个类继承自ClassLoader抽象类。
2. 复写它的findClass()方法。
3. 在findClass()方法中调用defineClass()。

ClassLoader cl = Test.class.getClassLoader();

System.out.println("ClassLoader is:"+cl.toString());

cl = int.class.getClassLoader();

System.out.println("ClassLoader is:"+cl.toString());

1. 一个AppClassLoader查找资源时，先看看缓存是否有，缓存有从缓存中获取，否则委托给父加载器。   
2. 递归，重复第1部的操作。   
3. 如果ExtClassLoader也没有加载过，则由Bootstrap ClassLoader出面，它首先查找缓存，如果没有找到的话，就去找自己的规定的路径下，也就是sun.mic.boot.class下面的路径。找到就返回，没有找到，让子加载器自己去找。   
4. Bootstrap ClassLoader如果没有查找成功，则ExtClassLoader自己在java.ext.dirs路径中去查找，查找成功就返回，查找不成功，再向下让子加载器找。   
5. ExtClassLoader查找不成功，AppClassLoader就自己查找，在java.class.path路径下查找。找到就返回。如果没有找到就让子类找，如果没有子类会怎么样？抛出各种异常。

# jvm内存分配。

       按照官方的说法：“Java 虚拟机具有一个堆，堆是运行时数据区域，所有类实例和数组的内存均从此处分配。堆是在 Java 虚拟机启动时创建的。”“在JVM中堆之外的内存称为非堆内存(Non-heap memory)”。可以看出JVM主要管理两种类型的内存：堆和非堆。简单来说堆就是Java代码可及的内存，是留给开发人员使用的；非堆就是JVM留给 自己用的，所以方法区、JVM内部处理或优化所需的内存(如JIT编译后的代码缓存)、每个类结构(如运行时常数池、字段和方法数据)以及方法和构造方法 的代码都在非堆内存中。

* 方法栈&本地方法栈:  
  线程创建时产生,方法执行时生成栈帧
* 方法区  
  存储类的元数据信息 常量等
* 堆  
  java代码中所有的new操作
* native Memory(C heap)  
  Direct Bytebuffer JNI Compile GC;

JVM初始分配的内存由-Xms指定，默认是物理内存的1/64；JVM最大分配的内存由-Xmx指 定，默认是物理内存的1/4。默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制。因此服务器一般设置-Xms、-Xmx相等以避免在每次GC 后调整堆的大小。对象的堆内存由称为垃圾回收器的自动内存管理系统回收。

**String s = "abc";**  
创建过程分析：在class文件被JVM装载到内存中，JVM会创建一块String Pool（String缓冲池）。当执行String s = “abc”;时，JVM首先在String Pool中查看是否存在字符串对象“abc”（如何查看呢？用equals()方法判断），如果已存在该对象，则不用创建新的字符串对象“abc”，而直接使用String Pool中已存在的对象“abc”，然后将引用s指向该对象；如果不存在该对象，则先在String Pool中创建一个新的字符串对象“abc”，然后将引用s指向String Pool中创建的新对象。

**String s = new String("abc");**  
创建过程分析：当执行String s = new String(“abc”);时，JVM首先在String Pool中查看是否存在字符串对象“abc”，如果不存在该对象，则先在String Pool中创建一个新的字符串对象“abc”，然后执行new String(“abc”)构造方法，在Heap里又创建一个新的字符串对象“abc”（new出来的对象都放在Heap里面），并将引用s指向Heap中创建的新对象；如果已存在该对象，则不用创建新的字符串对象“abc”，而直接使用String Pool中已存在的对象“abc”， 然后执行new String(“abc”)构造方法，在Heap里又创建一个新的字符串对象“abc”，并将引用s指向Heap中创建的新对象。   
注意：使用new String（“”）创建的字符串对象时，会在运行期创建新对象存储到Heap中。因此，new String（“abc”）创建字符串对象时，会创建2个对象，编译期在String Pool中创建一个，运行时Heap中创建一个。

1. **public** **class** TestString{
2. **public** **static** **void** main(String args[]){
3. String s1 = **new** String("abc");//语句1
4. String s2 = "abc";//语句2
5. String s3 = **new** String("abc");//语句3
7. System.out.println(s1 == s2);//语句4
8. System.out.println(s1 == s3);//语句5
9. System.out.println(s2 == s3);//语句6
11. System.out.println(s1 == s1.intern());//语句7
12. System.out.println(s2 == s2.intern());//语句8
13. System.out.println(s1.intern() == s2.intern());//语句9
15. String hello = "hello";//语句10
16. String hel = "hel";//语句11
17. String lo = "lo";//语句12
19. System.out.println(hello == "hello");//语句13
20. System.out.println(hello == "hel" + "lo");//语句14
21. System.out.println(hello == "hel" + lo);//语句15
22. System.out.println(hello == hel + lo);//语句16
23. }
24. }

**问题1：当执行完语句(1)时，在内存里面生成几个对象?它们是什么?在什么地方?**  
--->当执行完语句(1)时，在内存里面创建了两个对象，它们的内容分别都是abc，分别在String Pool(常量池)和Heap(堆)里。

其字符串的创建过程如下：首先在String Pool里面查找查找是否有 "abc"，如果有就直接使用，但这是本程序的第一条语句，故不存在一个对象"abc"，所以要在String Pool中生成一个对象"abc"，接下来，执行new String("abc")构造方法，new出来的对象都放在Heap里面。在Heap里又创建了一个"abc"的对象。这时内存里就有两个对象了，一个在String Pool 里面，一个在Heap里面。   
  
**问题2：当执行完语句(2)时，在内存里面一共有几个对象?它们是什么?在什么地方?**  
当执行完语句(2)时，在内存里面一个对象也没有创建。当我们定义语句(2)的时候，如果我们用字符串的常量值(字面值)给s2赋值的话，那么首先JVM还是从String Pool里面去查找有没有内容为abc的这样一个对象存在，我们发现当我们执行完语句(1)的时候，StringPool里面已经存在了内容为abc的对象，那么就不会再在String Pool里面去生成内容为abc的字符串对象了。而是会使用已经存在String Pool里面的内容为abc的字符串对象，并且会将s2这个引用指向String Pool里面的内容为abc的字符串对象，s2存放的是String Pool里面的内容为abc的字符串对像的地址。也就是说当你使用String s2 = "abc"，即使用字符串常量("abc")给定义的引用(str2)赋值的话，那么它首先是在String Pool里面去找有没有内容为abc的字符串对象存在，如果有的话，就不用创建新的对象，直接引用String Pool里面已经存在的对象；如果没有的话，就在 String Pool里面去创建一个新的对象，接着将引用指向这个新创建的对象。所以，当执行完语句(2)时内存里面一共有2个对象，它们的内容分别都是abc，在String Pool里面一个内容abc的对象，在Heap里面有一个内容为abc的对象。   
  
**问题3：当执行完语句(3)时，在内存里面一共有几个对象?它们是什么?在什么地方?**  
当执行完语句(3)时，其执行过程是这样的：它首先在String Pool里面去查找有没有内容为abc的字符串对象存在，发现有这个对象存在，它就不去创建 一个新的对象。接着执行new...，只要在java里面有关键字new存在，不管内容是否相同，都表示它将生成一个新的对象，new多少次，就生成多少个对象，而且新生成的对象都是在Heap里面，所以它会在Heap里面生成一个内容为abc的对象，并且将它的地址赋给了引用s3，s3就指向刚在Heap里面生成的内容为abc的对象。所以，当执行完语句(3)时，内存里面一共有3个对象，其中包含了在String Pool里面一个内容为abc的字符串对象和在Heap里面包含了两个内容为abc的字符串对象。   
  
**问题4：当执行完语句(4)(5)(6)后，它们的结果分别是什么?**  
在java里面，对象用"=="永远比较的是两个对象的内存地址，换句话说，是比较"=="左右两边的两个引用是否指向同一个对象。对于java里面的8种原生数据类型来说，"=="比较的是它们的字面值是不是一样的；对应用类型来说，比较的是它们的内存地址是不是一样的。在语句(1)(2)(3)中，由于s1、s2、s3指向不同的对象，它们的内存地址就不一样，因此可以说当执行完语句(4)(5)(6),它们返回的结果都是false。   
 **问题5：当执行完语句(7)(8)(9)后，它们的结果分别是什么?**  
首先，s1这个对象指向的是堆中第一次new...生成的对象，当调用 intern 方法时，如果String Pool已经包含一个等于此 String 对象的字符串(该对象由equals(Object)方法确定），则返回指向String Pool中的字符串对象的引用。因为String Pool中有内容为abc的对象，所以s1.intern()返回的是String Pool中的内容为abc的字符串对象的内存地址，而s1却是指向Heap上内容为abc的字符串对象的引用。因而，两个引用指向的对象不同，所以，s1 == s1.intern() 为false，即语句(7)结果为false。   
对于s2.intern()，它还是会首先检查String Pool中是否有内容为abc的对象，发现有，则将String Pool中内容为abc的对象的地址赋给s2.intern()方法的返回值。因为s2和s2.intern()方法的返回值指向的是同一个对象，所以，s2 == s2.intern()的结果为true,，即语句(8)结果为true。   
对于s1.intern()，它首先检查String Pool中是否有内容为abc的对象，发现有，则将String Pool中内容为abc的对象的赋给s1.intern()方法的返回值。对于s2.intern(),首先检查String Pool中是否有内容为abc的对象，发现有，则将String Pool中内容为abc的对象的地址赋给s2.intern()方法的返回值。因为两者返回的地址都指向同一个对象，所以，s1.intern() == s2.intern()的结果为true,，即是语句(9)结果为true。   
因此，当执行完语句(7)(8)(9)后，它们的结果分别是false、true、true。   
  
**问题6：当执行完语句(13)(14) (15)(16)后，它们的结果分别是什么?**  
hello == "hello"引用hello指向的对象就是String Pool中的“hello”，即语句(13)的结果为true。   
hello == "hel" + "lo"当加号两边都是常量值时，就会组成一个新的常量值"hello"在String Pool里面，如果String Pool已经有相同内容的就不会再创建，则直接返回String Pool里面的内容为"hello"的字符串对象的内存地址，所以，hello == "hel" + "lo"结果为true。   
hello =="hel" + lo 当加号两边有一个不是常量值，会在堆里面创建一个新的"hello"对象，一个在String Pool中，一个在Heap中，故输出false 。   
hel + lo 同上，输出false。   
因此，当执行完语句(7)(8)(9)后，它们的结果分别是true、true、false、false。

# 多线程的包，blockQuence，current

我们都知道，在JDK1.5之前，[Java](http://lib.csdn.net/base/java)中要进行业务并发时，通常需要有程序员独立完成代码实现，当然也有一些开源的框架提供了这些功能，但是这些依然没有JDK自带的功能使用起来方便。而当针对高质量Java多线程并发程序设计时,为防止死蹦等现象的出现，比如使用java之前的wait()、notify()和synchronized等，每每需要考虑性能、死锁、公平性、资源管理以及如何避免线程安全性方面带来的危害等诸多因素，往往会采用一些较为复杂的安全策略，加重了程序员的开发负担.万幸的是，在JDK1.5出现之后，Sun大神（Doug Lea）终于为我们这些可怜的小程序员推出了java.util.concurrent工具包以简化并发完成。开发者们借助于此，将有效的减少竞争条件（race conditions）和死锁线程。concurrent包很好的解决了这些问题，为我们提供了更实用的并发程序模型。

Executor                  ：具体Runnable任务的执行者。  
ExecutorService           ：一个线程池管理者，其实现类有多种，我会介绍一部分。我们能把Runnable,Callable提交到池中让其调度。  
Semaphore                 ：一个计数信号量  
ReentrantLock             ：一个可重入的互斥锁定 Lock，功能类似synchronized，但要强大的多。  
Future                    ：是与Runnable,Callable进行交互的接口，比如一个线程执行结束后取返回的结果等等，还提供了cancel终止线程。  
BlockingQueue             ：阻塞队列。  
CompletionService         : ExecutorService的扩展，可以获得线程执行结果的  
CountDownLatch            ：一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。   
CyclicBarrier             ：一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点   
Future                    ：Future 表示异步计算的结果。  
ScheduledExecutorService ：一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

接下来逐一介绍

BlockingQueue  
支持两个附加操作的 Queue，这两个操作是：检索元素时等待队列变为非空，以及存储元素时等待空间变得可用。

BlockingQueue 不接受 null 元素。试图 add、put 或 offer 一个 null 元素时，某些实现会抛出 NullPointerException。null 被用作指示 poll 操作失败的警戒值。

BlockingQueue 可以是限定容量的。它在任意给定时间都可以有一个 remainingCapacity，超出此容量，便无法无阻塞地 put 额外的元素。  
没有任何内部容量约束的 BlockingQueue 总是报告 Integer.MAX\_VALUE 的剩余容量。

BlockingQueue 实现主要用于生产者-使用者队列，但它另外还支持 Collection 接口。因此，举例来说，使用 remove(x) 从队列中移除任意一个元素是有可能的。  
然而，这种操作通常不 会有效执行，只能有计划地偶尔使用，比如在取消排队信息时。

BlockingQueue 实现是线程安全的。所有排队方法都可以使用内部锁定或其他形式的并发控制来自动达到它们的目的。  
然而，大量的 Collection 操作（addAll、containsAll、retainAll 和 removeAll）没有 必要自动执行，除非在实现中特别说明。  
因此，举例来说，在只添加了 c 中的一些元素后，addAll(c) 有可能失败（抛出一个异常）。

BlockingQueue 实质上不 支持使用任何一种“close”或“shutdown”操作来指示不再添加任何项。  
这种功能的需求和使用有依赖于实现的倾向。例如，一种常用的策略是：对于生产者，插入特殊的 end-of-stream 或 poison 对象，并根据使用者获取这些对象的时间来对它们进行解释。

下面的例子演示了这个阻塞队列的基本功能。

import java.util.concurrent.BlockingQueue;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.LinkedBlockingQueue;

public class MyBlockingQueue extends Thread {  
public static BlockingQueue<String> queue = new LinkedBlockingQueue<String>(3);

private int index;

public MyBlockingQueue(int i) {  
   this.index = i;  
}

public void run() {  
   try {  
    queue.put(String.valueOf(this.index));  
    System.out.println("{" + this.index + "} in queue!");  
   } catch (Exception e) {  
    e.printStackTrace();  
   }  
}

public static void main(String args[]) {  
   ExecutorService service = Executors.newCachedThreadPool();  
   for (int i = 0; i < 10; i++) {  
    service.submit(new MyBlockingQueue(i));  
   }  
   Thread thread = new Thread() {  
    public void run() {  
     try {  
      while (true) {  
       Thread.sleep((int) (Math.random() \* 1000));  
       if(MyBlockingQueue.queue.isEmpty())  
        break;  
       String str = MyBlockingQueue.queue.take();  
       System.out.println(str + " has take!");  
      }  
     } catch (Exception e) {  
      e.printStackTrace();  
     }  
    }  
   };  
   service.submit(thread);  
   service.shutdown();  
}  
}

Future

Future 表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。  
计算完成后只能使用 get 方法来检索结果，如有必要，计算完成前可以阻塞此方法。取消则由 cancel 方法来执行。  
还提供了其他方法，以确定任务是正常完成还是被取消了。一旦计算完成，就不能再取消计算。  
如果为了可取消性而使用 Future但又不提供可用的结果，则可以声明 Future<?> 形式类型、并返回 null 作为基础任务的结果。

这个我们在前面CompletionService已经看到了，这个Future的功能，而且这个可以在提交线程的时候被指定为一个返回对象的。

ScheduledExecutorService

一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

schedule 方法使用各种延迟创建任务，并返回一个可用于取消或检查执行的任务对象。scheduleAtFixedRate 和 scheduleWithFixedDelay 方法创建并执行某些在取消前一直定期运行的任务。

用 Executor.execute(java.lang.Runnable) 和 ExecutorService 的 submit 方法所提交的命令，通过所请求的 0 延迟进行安排。  
schedule 方法中允许出现 0 和负数延迟（但不是周期），并将这些视为一种立即执行的请求。

所有的 schedule 方法都接受相对 延迟和周期作为参数，而不是绝对的时间或日期。将以 Date 所表示的绝对时间转换成要求的形式很容易。  
例如，要安排在某个以后的日期运行，可以使用：schedule(task, date.getTime() - System.currentTimeMillis(), TimeUnit.MILLISECONDS)。  
但是要注意，由于网络时间同步协议、时钟漂移或其他因素的存在，因此相对延迟的期满日期不必与启用任务的当前 Date 相符。  
Executors 类为此包中所提供的 ScheduledExecutorService 实现提供了便捷的工厂方法。

一下的例子也是网上比较流行的。

import static java.util.concurrent.TimeUnit.SECONDS;  
import java.util.Date;  
import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.ScheduledFuture;

public class TestScheduledThread {  
public static void main(String[] args) {  
   final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);  
   final Runnable beeper = new Runnable() {  
    int count = 0;

    public void run() {  
     System.out.println(new Date() + " beep " + (++count));  
    }  
   };  
   // 1秒钟后运行，并每隔2秒运行一次  
   final ScheduledFuture beeperHandle = scheduler.scheduleAtFixedRate(beeper, 1, 2, SECONDS);  
   // 2秒钟后运行，并每次在上次任务运行完后等待5秒后重新运行  
   final ScheduledFuture beeperHandle2 = scheduler.scheduleWithFixedDelay(beeper, 2, 5, SECONDS);  
   // 30秒后结束关闭任务，并且关闭Scheduler  
   scheduler.schedule(new Runnable() {  
    public void run() {  
     beeperHandle.cancel(true);  
     beeperHandle2.cancel(true);  
     scheduler.shutdown();  
    }  
   }, 30, SECONDS);  
}  
}

# aop底层实现。

Spring AOP**采用动态代理机制和字节码生成技术实现**。与第一代AspectJ采用编译器将横切逻辑织入目标对象不同，动态代理机制和字节码生成都是在

运行期间为目标对象生成一个代理对象，而将横切逻辑织入到这个代理对象中，系统最终使用的是织入了横切逻辑的代理对象，而不是真正的目标对象。

默认情况下，如果Spring AOP发现目标对象实现了相应Interface，则采用动态代理机制为其生成代理对象实例。而如果目标对象没有实现任何Interface

，Spring AOP会尝试使用CGLIB(Code Generation Library)这种动态字节码生成类库，为目标对象生成动态的代理对象实例。

# redis备份机制。

Redis 提供了多种不同级别的持久化方式：

* RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。
* AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小。
* Redis 还可以同时使用 AOF 持久化和 RDB 持久化。在这种情况下，当 Redis 重启时，它会优先使用 AOF 文件来还原数据集，因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。
* 你甚至可以关闭持久化功能，让数据只在服务器运行时存在。

## 备份 Redis 数据

在阅读这个小节前，先将下面这句话铭记于心：一定要备份你的数据库！

磁盘故障，节点失效，诸如此类的问题都可能让你的数据消失不见，不进行备份是非常危险的。

RDB:关系数据库

Redis 对于数据备份是非常友好的，因为你可以在服务器运行的时候对 RDB 文件进行复制：RDB 文件一旦被创建，就不会进行任何修改。当服务器要创建一个新的 RDB 文件时，它先将文件的内容保存在一个临时文件里面，当临时文件写入完毕时，程序才使用 rename(2) 原子地用临时文件替换原来的 RDB 文件。

这也就是说，无论何时，复制 RDB 文件都是绝对安全的。

以下是我们的建议：

* 创建一个定期任务（cron job），每小时将一个 RDB 文件备份到一个文件夹，并且每天将一个 RDB 文件备份到另一个文件夹。
* 确保快照的备份都带有相应的日期和时间信息，每次执行定期任务脚本时，使用 find 命令来删除过期的快照：比如说，你可以保留最近 48 小时内的每小时快照，还可以保留最近一两个月的每日快照。
* 至少每天一次，将 RDB 备份到你的数据中心之外，或者至少是备份到你运行 Redis 服务器的物理机器之外。

## 容灾备份

Redis 的容灾备份基本上就是对数据进行备份，并将这些备份传送到多个不同的外部数据中心。

容灾备份可以在 Redis 运行并产生快照的主数据中心发生严重的问题时，仍然让数据处于安全状态。

因为很多 Redis 用户都是创业者，他们没有大把大把的钱可以浪费，所以下面介绍的都是一些实用又便宜的容债备份方法：

* Amazon S3 ，以及其他类似 S3 的服务，是一个构建灾难备份系统的好地方。最简单的方法就是将你的每小时或者每日 RDB 备份加密并传送到 S3 。对数据的加密可以通过 gpg -c 命令来完成（对称加密模式）。记得把你的密码放到几个不同的、安全的地方去（比如你可以把密码复制给你组织里最重要的人物）。同时使用多个储存服务来保存数据文件，可以提升数据的安全性。
* 传送快照可以使用 SCP 来完成（SSH 的组件）。以下是简单并且安全的传送方法：买一个离你的数据中心非常远的 VPS ，装上 SSH ，创建一个无口令的 SSH 客户端 key ，并将这个 key 添加到 VPS 的 authorized\_keys 文件中，这样就可以向这个 VPS 传送快照备份文件了。为了达到最好的数据安全性，至少要从两个不同的提供商那里各购买一个 VPS 来进行数据容灾备份。

需要注意的是，这类容灾系统如果没有小心地进行处理的话，是很容易失效的。

最低限度下，你应该在文件传送完毕之后，检查所传送备份文件的体积和原始快照文件的体积是否相同。如果你使用的是 VPS ，那么还可以通过比对文件的 SHA1 校验和来确认文件是否传送完整。

另外，你还需要一个独立的警报系统，让它在负责传送备份文件的传送器（transfer）失灵时通知你。

# 隔离级别，事务的传播特性。

[**spring**](http://lib.csdn.net/base/javaee)**事务一个被讹传很广说法是：一个事务方法不应该调用另一个事务方法，否则将产生两个事务。结果造成开发人员在设计事务方法时束手束脚，生怕一不小心就踩到地雷。**

其实这种是不认识 Spring 事务传播机制而造成的误解，Spring 对事务控制的支持统一在 TransactionDefinition 类中描述，该类有以下几个重要的接口方法：

* int getPropagationBehavior()：事务的传播行为
* int getIsolationLevel()：事务的隔离级别
* int getTimeout()：事务的过期时间
* boolean isReadOnly()：事务的读写特性。

很明显，除了事务的传播行为外，事务的其它特性 Spring 是借助底层资源的功能来完成的，Spring 无非只充当个代理的角色。但是事务的传播行为却是 Spring 凭借自身的框架提供的功能，是 Spring 提供给开发者最珍贵的礼物，讹传的说法玷污了 Spring 事务框架最美丽的光环。

所谓事务传播行为就是多个事务方法相互调用时，事务如何在这些方法间传播。Spring 支持 7 种事务传播行为：

* PROPAGATION\_REQUIRED 如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。
* PROPAGATION\_SUPPORTS 支持当前事务，如果当前没有事务，就以非事务方式执行。
* PROPAGATION\_MANDATORY 使用当前的事务，如果当前没有事务，就抛出异常。
* PROPAGATION\_REQUIRES\_NEW 新建事务，如果当前存在事务，把当前事务挂起。
* PROPAGATION\_NOT\_SUPPORTED 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
* PROPAGATION\_NEVER 以非事务方式执行，如果当前存在事务，则抛出异常。
* PROPAGATION\_NESTED 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与 PROPAGATION\_REQUIRED 类似的操作。

Spring 默认的事务传播行为是 PROPAGATION\_REQUIRED，它适合于绝大多数的情况。假设 ServiveX#methodX() 都工作在事务环境下（即都被 Spring 事务增强了），假设程序中存在如下的调用链：Service1#method1()->Service2#method2()->Service3#method3()，那么这 3 个服务类的 3 个方法通过 Spring 的事务传播机制都工作在同一个事务中。

4、Spring事务的隔离级别  
  
1、 ISOLATION\_DEFAULT: 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别。  
2、ISOLATION\_READ\_UNCOMMITTED:这是事务最低的隔离级别，它允许另外一个事务可以看到这个事务未提交的数据。  
3、ISOLATION\_READ\_COMMITTED：保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。  
4、ISOLATION\_REPEATALBE\_READ: 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻想读。它除了保证一个事务不能读取另外一个事务未提交的数据外，还保证了避免下面的情况产生（不可重复读）。  
5、ISOLATION\_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不课重复读外，还避免了幻想读。

本地事务

[**数据库**](http://lib.csdn.net/base/mysql)事务，默认事务为自动提交，因此如果一个业务逻辑类中有多次数据库操作将无法保证事务的一致性。

Spring事务

       对本地事务操作的一次封装，相当于把使用JDBC代码开启、提交、回滚事务进行了封装。

上述两个概念会在demo中用到，以方便大家理解代码。

传播特性

       该特性是保证事务是否开启，业务逻辑是否使用同一个事务的保证。当事务在传播过程中会受其影响。其传播特性包括：

  1、Propagation.REQUIRED

方法被调用时自动开启事务，在事务范围内使用则使用同一个事务，否则开启新事务。

  2、Propagation.REQUIRES\_NEW

无论何时自身都会开启事务

  3、Propagation.SUPPORTS

自身不会开启事务，在事务范围内则使用相同事务，否则不使用事务

  4、Propagation.NOT\_SUPPORTED

自身不会开启事务，在事务范围内使用挂起事务，运行完毕恢复事务

  5、Propagation.MANDATORY

自身不开启事务，必须在事务环境使用否则报错

  6、Propagation.NEVER

自身不会开启事务，在事务范围使用抛出异常

  7、Propagation.NESTED

如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按TransactionDefinition.PROPAGATION\_REQUIRED 属性执行。需要JDBC3.0以上支持。

**1.脏读：**  
脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。

**2.不可重复读：**  
是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。（即不能读到相同的数据内容）  
例如，一个编辑人员两次读取同一文档，但在两次读取之间，作者重写了该文档。当编辑人员第二次读取文档时，文档已更改。原始读取不可重复。如果只有在作者全部完成编写后编辑人员才可以读取文档，则可以避免该问题。

**3.幻读:**  
是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象  
发生了幻觉一样。  
例如，一个编辑人员更改作者提交的文档，但当生产部门将其更改内容合并到该文档的主复本时，发现作者已将未编辑的新材料添加到该文档中。如果在编辑人员和生产部门完成对原始文档的处理之前，任何人都不能将新材料添加到文档中，则可以避免该问题。

数据库事务的隔离级别有4个，由低到高依次为Read uncommitted 、Read committed 、Repeatable read 、Serializable ，这四个级别可以逐个解决脏读 、不可重复读 、幻读 这几类问题。

√: 可能出现    ×: 不会出现

|  |  |  |  |
| --- | --- | --- | --- |
|  | 脏读 | 不可重复读 | 幻读 |
| Read uncommitted | √ | √ | √ |
| Read committed | × | √ | √ |
| Repeatable read | × | × | √ |
| Serializable | × | × | × |

注意：我们讨论隔离级别的场景，主要是在多个事务并发 的情况下，因此，接下来的讲解都围绕事务并发。

##### Read uncommitted 读未提交

公司发工资了，领导把5000元打到singo的账号上，但是该事务并未提交，而singo正好去查看账户，发现工资已经到账，是5000元整，非常高 兴。可是不幸的是，领导发现发给singo的工资金额不对，是2000元，于是迅速回滚了事务，修改金额后，将事务提交，最后singo实际的工资只有 2000元，singo空欢喜一场。

出现上述情况，即我们所说的脏读 ，两个并发的事务，“事务A：领导给singo发工资”、“事务B：singo查询工资账户”，事务B读取了事务A尚未提交的数据。

当隔离级别设置为Read uncommitted 时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

##### Read committed 读提交

singo拿着工资卡去消费，系统读取到卡里确实有2000元，而此时她的老婆也正好在网上转账，把singo工资卡的2000元转到另一账户，并在 singo之前提交了事务，当singo扣款时，系统检查到singo的工资卡已经没有钱，扣款失败，singo十分纳闷，明明卡里有钱，为 何......

出现上述情况，即我们所说的不可重复读 ，两个并发的事务，“事务A：singo消费”、“事务B：singo的老婆网上转账”，事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变。

当隔离级别设置为Read committed 时，避免了脏读，但是可能会造成不可重复读。

大多数数据库的默认级别就是Read committed，比如Sql Server , [**Oracle**](http://lib.csdn.net/base/oracle)。如何解决不可重复读这一问题，请看下一个隔离级别。

##### Repeatable read 重复读

当隔离级别设置为Repeatable read 时，可以避免不可重复读。当singo拿着工资卡去消费时，一旦系统开始读取工资卡信息（即事务开始），singo的老婆就不可能对该记录进行修改，也就是singo的老婆不能在此时转账。

虽然Repeatable read避免了不可重复读，但还有可能出现幻读 。

singo的老婆工作在银行部门，她时常通过银行内部系统查看singo的信用卡消费记录。有一天，她正在查询到singo当月信用卡的总消费金额 （select sum(amount) from transaction where month = 本月）为80元，而singo此时正好在外面胡吃海塞后在收银台买单，消费1000元，即新增了一条1000元的消费记录（insert transaction ... ），并提交了事务，随后singo的老婆将singo当月信用卡消费的明细打印到A4纸上，却发现消费总额为1080元，singo的老婆很诧异，以为出 现了幻觉，幻读就这样产生了。

注：[**MySQL**](http://lib.csdn.net/base/mysql)的默认隔离级别就是Repeatable read。

##### Serializable 序列化

Serializable 是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

**实例Demo**

Propagation.REQUIRED

测试入口代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. <span style="font-size:14px">//会开启事务，在事务范围内使用则使用同一个事务，否则开启新事务
2. @Test
3. **public** **void** testRequires(){
4. sService.addStudent();
5. }</span>

Service代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.REQUIRED)
2. **public** **void** addStudent(){
3. <span style="white-space:pre">  </span>String sql = "insert into student(name) values('st0')";
4. jdbcTemplate.execute(sql);
5. tService.addTeacher();
6. **throw** **new** RuntimeException();
7. }

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.REQUIRES)
2. **public** **void** addTeacher(){
3. String sql = "insert into teacher(name) values ('t5')";
4. jdbcTemplate.execute(sql);
5. }

经测试无论在tService还是sService如果不抛出异常，那么数据提交成功，如果抛出异常，数据提交失败。这说明tService和sService使用的是同一个事务，并且只要方法被调用就开启事务。

Propagation.REQUIRES\_NEW

测试入口代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. //无论何时自身都会开启事务
2. @Test
3. **public** **void** testRequiresNew(){
4. sService.addStudent5();
5. }

Service代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.REQUIRES\_NEW)
2. **public** **void** addStudent5(){
3. String sql = "insert into student(name) values('st5')";
4. jdbcTemplate.execute(sql);
5. tService.addTeacher5();
6. **throw** **new** RuntimeException();
7. }

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.REQUIRES\_NEW)
2. **public** **void** addTeacher5(){
3. String sql = "insert into teacher(name) values ('t5')";
4. jdbcTemplate.execute(sql);
5. }

经测试如果在addStudent5中抛出异常，学生数据不能正确提交，教师信息被正确提交。说明sService和tService是在两个独立的事务中运行，并且只要方法被调用就开启事务。

Propagation.SUPPORTS

测试入口代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. //自身不会开启事务，在事务范围内则使用相同事务，否则不使用事务
2. @Test
3. **public** **void** testSupport(){
4. sService.addStudent6();
5. }

Service代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.SUPPORTS)
2. **public** **void** addStudent6(){
3. String sql = "insert into student(name) values('st6')";
4. jdbcTemplate.execute(sql);
5. tService.addTeacher6();
6. **throw** **new** RuntimeException();
7. }

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.SUPPORTS)
2. **public** **void** addTeacher6(){
3. String sql = "insert into teacher(name) values ('t6')";
4. jdbcTemplate.execute(sql);
5. }

经测试如果在addStudent6中抛出异常，学生数据和教师数据都被正确提交。说明sService和tService没有被spring管理和开启事务，而是使用了本地事务，由于本地事务默认自动提交因此数据都提交成功，但它们使用的却不是同一个事务，一旦出现异常将导致数据的不一致。

Propagation.NOT\_SUPPORTED

测试入口代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. //自身不会开启事务，在事务范围内使用挂起事务，运行完毕恢复事务
2. @Test
3. **public** **void** testNotSupport(){
4. sService.addStudent4();
5. }

Service代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.NOT\_SUPPORTED)
2. **public** **void** addStudent4(){
3. String sql = "insert into student(name) values('st4')";
4. jdbcTemplate.execute(sql);
5. **throw** **new** RuntimeException();
6. }

经测试如果在addStudent4中抛出异常，学生数据正确提交。说明sService没有被spring管理和开启事务，而是使用了本地事务，由于本地事务默认自动提交因此数据都提交成功。

测试入口代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. //自身不会开启事务，在事务范围内使用挂起事务，运行完毕恢复事务
2. @Test
3. **public** **void** testNotSupport1(){
4. sService.addStudent();
5. }

Service代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.REQUIRED)
2. **public** **void** addStudent(){
3. String sql = "insert into student(name) values('st0')";
4. jdbcTemplate.execute(sql);
5. tService.addTeacher4();
6. }

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.NOT\_SUPPORTED)
2. **public** **void** addTeacher4(){
3. String sql = "insert into teacher(name) values ('t4')";
4. jdbcTemplate.execute(sql);
5. **throw** **new** RuntimeException();
6. }

经测试如果在addTeacher4中抛出异常，学生数据提交失败，教师数据提交成功。说明sService开启了事务，tService没有开启事务，而是使用了本地事务。

Propagation.MANDATORY

测试入口代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. //自身不开启事务，必须在事务环境使用否则报错
2. @Test
3. **public** **void** testMandatory(){
4. sService.addStudent1();
5. }

Service代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.MANDATORY)
2. **public** **void** addStudent1(){
3. String sql = "insert into student(name) values('st1')";
4. jdbcTemplate.execute(sql);
5. }

经测试代码报错。

org.springframework.transaction.IllegalTransactionStateException: No existing transaction found for transaction marked with propagation 'mandatory'，没有找到事务环境。

Propagation.NEVER

测试入口代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. <span style="font-size:14px">//自身不会开启事务，在事务范围使用抛出异常
2. @Test
3. **public** **void** testNever(){
4. sService.addStudent();
5. }</span>

Service代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. <span style="font-size:14px">@Transactional(propagation = Propagation.REQUIRED)
2. **public** **void** addStudent(){
3. String sql = "insert into student(name) values('st0')";
4. jdbcTemplate.execute(sql);
5. tService.addTeacher3();
6. }</span>

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. <span style="font-size:14px">@Transactional(propagation = Propagation.NEVER)
2. **public** **void** addTeacher3(){
3. String sql = "insert into teacher(name) values ('t3')";
4. jdbcTemplate.execute(sql);
5. }</span><span style="font-size:18px">
6. </span>

经测试代码报错，由于sService开启了事务，当调用sService方法时由于其传播特性为never，因此报存在事务错误。

org.springframework.transaction.IllegalTransactionStateException: Existing transaction found for transaction marked with propagation 'never'

Propagation.NESTED

测试入口代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. //如果没有事务环境其特性同Propagation.REQUIRED,否则嵌套运行事务
2. @Test
3. **public** **void** testNested(){
4. sService.addStudent2();
5. }

Service代码

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.NESTED)
2. **public** **void** addStudent2(){
3. String sql = "insert into student(name) values('st2')";
4. jdbcTemplate.execute(sql);
5. tService.addTeacher2();
6. **throw** **new** RuntimeException();
7. }

**[java]** [view plain](http://blog.csdn.net/liovey/article/details/14149137) [copy](http://blog.csdn.net/liovey/article/details/14149137)

1. @Transactional(propagation = Propagation.NESTED)
2. **public** **void** addTeacher2(){
3. String sql = "insert into teacher(name) values ('t2')";
4. jdbcTemplate.execute(sql);
5. }

# 索引内部，建索引情况，重复率

单例模式和多例模式

Spring框架里的bean，或者说组件，获取实例的时候都是默认的单例模式，这是在多线程开发的时候要尤其注意的地方。

单例模式的意思就是只有一个实例。单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

当多用户同时请求一个服务时，容器会给每一个请求分配一个线程，这是多个线程会并发执行该请求多对应的业务逻辑（成员方法），此时就要注意了，如果该处理逻辑中有对该单列状态的修改（体现为该单列的成员属性），则必须考虑线程同步问题

同步机制的比较　　ThreadLocal和线程同步机制相比有什么优势呢？ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。

　　在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

　　而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

　　由于ThreadLocal中可以持有任何类型的对象，低版本JDK所提供的get()返回的是Object对象，需要强制类型转换。但JDK 5.0通过泛型很好的解决了这个问题，在一定程度地简化ThreadLocal的使用

　概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

　　Spring使用ThreadLocal解决线程安全问题

　　我们知道在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域。就是因为Spring对一些Bean（如RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder等）中非线程安全状态采用ThreadLocal进行处理，让它们也成为线程安全的状态，因为有状态的Bean就可以在多线程中共享了。

linux grep前后 统计pvp 统计后面的字段

为什么cglib代理接口

 CGLIB是一个强大的高性能的代码生成包。

      1>它广泛的被许多AOP的框架使用，例如：[**spring**](http://lib.csdn.net/base/javaee) AOP和dynaop，为他们提供方法的interception（拦截）；

       2>[**hibernate**](http://lib.csdn.net/base/javaee)使用CGLIB来代理单端single-ended(多对一和一对一)关联（对集合的延迟抓取，是采用其他机制实现的）；

       3>EasyMock和jMock是通过使用模仿（moke）对象来[**测试**](http://lib.csdn.net/base/softwaretest)[**Java**](http://lib.csdn.net/base/java)代码的包。

     它们都通过使用CGLIB来为那些没有接口的类创建模仿（moke）对象

同步除了sync还有别的？

1、使用 synchronized 关键字，这也是最原始的方法。代码如下

1. **synchronized**(anObject)
2. {
3. value = map.get(key);
4. }

JDK1.2 提供了 Collections.synchronizedMap(originMap) 方法，同步方式其实和上面这段代码相同。

 Map Collections.synchronizedMap(Map m)

这个方法返回一个同步的Map，这个Map封装了底层的HashMap的所有方法，使得底层的HashMap即使是在多线程的环境中也是安全的。

2、使用 JDK1.5 提供的锁（[**Java**](http://lib.csdn.net/base/java).util.concurrent.locks.Lock）。代码如下

1. lock.lock();
2. value = map.get(key);
3. lock.unlock();

3、实际应用中，可能多数操作都是读操作，写操作较少。针对这种情况，可以使用 JDK1.5 提供的读写锁（java.util.concurrent.locks.ReadWriteLock）。代码如下

1. rwlock.readLock().lock();
2. value = map.get(key);
3. rwlock.readLock().unlock();    
   这样两个读操作可以同时进行，理论上效率会比方法 2 高。

4、使用 JDK1.5 提供的 java.util.concurrent.ConcurrentHashMap 类。该类将 Map 的存储空间分为若干块，每块拥有自己的锁，大大减少了多个线程争夺同一个锁的情况。代码如下

1. value = map.get(key); //同步机制内置在 get 方法中

java5中新增了ConcurrentMap接口和它的一个实现类ConcurrentHashMap。

ConcurrentHashMap提供了和Hashtable以及SynchronizedMap中所不同的锁机制。Hashtable中采用的锁机制是一次锁住整个hash表，从而同一时刻只能由一个线程对其进行操作；而ConcurrentHashMap中则是一次锁住一个桶。ConcurrentHashMap默认将hash表分为16个桶，诸如get,put,remove等常用操作只锁当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有16个写线程执行，并发性能的提升是显而易见的。

上面说到的16个线程指的是写线程，而读操作大部分时候都不需要用到锁。只有在size等操作时才需要锁住整个hash表。

在迭代方面，ConcurrentHashMap使用了一种不同的迭代方式。在这种迭代方式中，当iterator被创建后集合再发生改变就不再是抛出ConcurrentModificationException，取而代之的是在改变时new新的数据从而不影响原有的数据 ，iterator完成后再将头指针替换为新的数据 ，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。

ROB和AOF优缺点

**RDB优点：**

1、紧凑易于备份，他就一个文件。

2、RDB可以最大化redis性能、父进程无需做任何操作只需要for一个子进程即可

3、恢复比AOF块

**RDB缺点：**

1、数据完整性：如果非常注重数据的完整性，那么RDB就不行，虽然他是一个point-in-time 的快照方式，但是在快照的过程中，redis重启了，那么在快照中的这些数据将会丢失

2、数据非常庞大后，非常耗CPU和时间，那么redis讲可能down掉1秒钟设置更长。

**AOF优点：**

1、 使用 AOF 持久化会让 Redis 变得非常耐久，AOF默认的每一秒追加一次也可以修改他的方式没执行一次命令追加一次，所以你最多丢失1秒钟的数据

2、 AOF 文件是一个只进行追加操作的日志文件（append only log）

3、 Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写

**AOF缺点：**

1、对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。

2、 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB

抢购实现

电商的秒杀和抢购，对我们来说，都不是一个陌生的东西。然而，从技术的角度来说，这对于Web系统是一个巨大的考验。当一个Web系统，在一秒钟内收到数以万计甚至更多请求时，系统的优化和稳定至关重要。这次我们会关注秒杀和抢购的技术实现和优化，同时，从技术层面揭开，为什么我们总是不容易抢到火车票的原因？

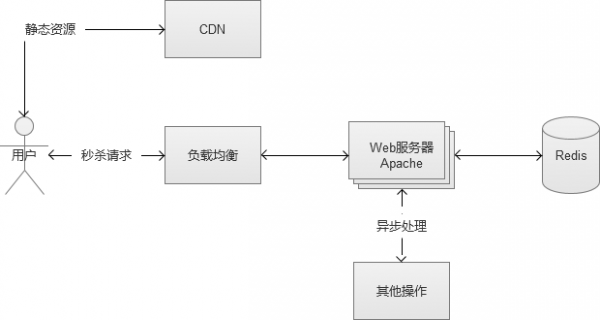
**一、大规模并发带来的挑战**

在过去的工作中，我曾经面对过5w每秒的高并发秒杀功能，在这个过程中，整个Web系统遇到了很多的问题和挑战。如果Web系统不做针对性的优化，会轻而易举地陷入到异常状态。我们现在一起来讨论下，优化的思路和方法哈。

**1. 请求接口的合理设计**

一个秒杀或者抢购页面，通常分为2个部分，一个是静态的HTML等内容，另一个就是参与秒杀的Web后台请求接口。

通常静态HTML等内容，是通过CDN的部署，一般压力不大，核心瓶颈实际上在后台请求接口上。这个后端接口，必须能够支持高并发请求，同时，非常重要的一点，必须尽可能“快”，在最短的时间里返回用户的请求结果。为了实现尽可能快这一点，接口的后端存储使用内存级别的操作会更好一点。仍然直接面向MySQL之类的存储是不合适的，如果有这种复杂业务的需求，都建议采用异步写入。



当然，也有一些秒杀和抢购采用“滞后反馈”，就是说秒杀当下不知道结果，一段时间后才可以从页面中看到用户是否秒杀成功。但是，这种属于“偷懒”行为，同时给用户的体验也不好，容易被用户认为是“暗箱操作”。

**2. 高并发的挑战：一定要“快”**

我们通常衡量一个Web系统的吞吐率的指标是QPS（Query Per Second，每秒处理请求数），解决每秒数万次的高并发场景，这个指标非常关键。举个例子，我们假设处理一个业务请求平均响应时间为100ms，同时，系统内有20台Apache的Web服务器，配置MaxClients为500个（表示Apache的最大连接数目）。

那么，我们的Web系统的理论峰值QPS为（理想化的计算方式）：

20\*500/0.1 = 100000 （10万QPS）

咦？我们的系统似乎很强大，1秒钟可以处理完10万的请求，5w/s的秒杀似乎是“纸老虎”哈。实际情况，当然没有这么理想。在高并发的实际场景下，机器都处于高负载的状态，在这个时候平均响应时间会被大大增加。

就Web服务器而言，Apache打开了越多的连接进程，CPU需要处理的上下文切换也越多，额外增加了CPU的消耗，然后就直接导致平均响应时间增加。因此上述的MaxClient数目，要根据CPU、内存等硬件因素综合考虑，绝对不是越多越好。可以通过Apache自带的abench来测试一下，取一个合适的值。然后，我们选择内存操作级别的存储的Redis，在高并发的状态下，存储的响应时间至关重要。网络带宽虽然也是一个因素，不过，这种请求数据包一般比较小，一般很少成为请求的瓶颈。负载均衡成为系统瓶颈的情况比较少，在这里不做讨论哈。

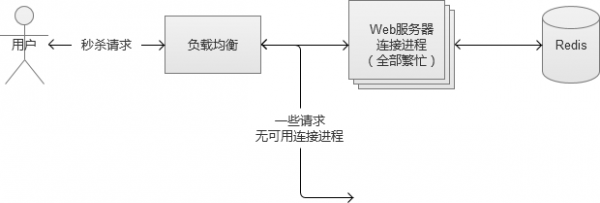
那么问题来了，假设我们的系统，在5w/s的高并发状态下，平均响应时间从100ms变为250ms（实际情况，甚至更多）：

20\*500/0.25 = 40000 （4万QPS）

于是，我们的系统剩下了4w的QPS，面对5w每秒的请求，中间相差了1w。

然后，这才是真正的恶梦开始。举个例子，高速路口，1秒钟来5部车，每秒通过5部车，高速路口运作正常。突然，这个路口1秒钟只能通过4部车，车流量仍然依旧，结果必定出现大塞车。（5条车道忽然变成4条车道的感觉）

同理，某一个秒内，20\*500个可用连接进程都在满负荷工作中，却仍然有1万个新来请求，没有连接进程可用，系统陷入到异常状态也是预期之内。



其实在正常的非高并发的业务场景中，也有类似的情况出现，某个业务请求接口出现问题，响应时间极慢，将整个Web请求响应时间拉得很长，逐渐将Web服务器的可用连接数占满，其他正常的业务请求，无连接进程可用。

更可怕的问题是，是用户的行为特点，系统越是不可用，用户的点击越频繁，恶性循环最终导致“雪崩”（其中一台Web机器挂了，导致流量分散到其他正常工作的机器上，再导致正常的机器也挂，然后恶性循环），将整个Web系统拖垮。

**3. 重启与过载保护**

如果系统发生“雪崩”，贸然重启服务，是无法解决问题的。最常见的现象是，启动起来后，立刻挂掉。这个时候，最好在入口层将流量拒绝，然后再将重启。如果是redis/memcache这种服务也挂了，重启的时候需要注意“预热”，并且很可能需要比较长的时间。

秒杀和抢购的场景，流量往往是超乎我们系统的准备和想象的。这个时候，过载保护是必要的。如果检测到系统满负载状态，拒绝请求也是一种保护措施。在前端设置过滤是最简单的方式，但是，这种做法是被用户“千夫所指”的行为。更合适一点的是，将过载保护设置在CGI入口层，快速将客户的直接请求返回。

**二、作弊的手段：进攻与防守**

秒杀和抢购收到了“海量”的请求，实际上里面的水分是很大的。不少用户，为了“抢“到商品，会使用“刷票工具”等类型的辅助工具，帮助他们发送尽可能多的请求到服务器。还有一部分高级用户，制作强大的自动请求脚本。这种做法的理由也很简单，就是在参与秒杀和抢购的请求中，自己的请求数目占比越多，成功的概率越高。

这些都是属于“作弊的手段”，不过，有“进攻”就有“防守”，这是一场没有硝烟的战斗哈。

**1. 同一个账号，一次性发出多个请求**

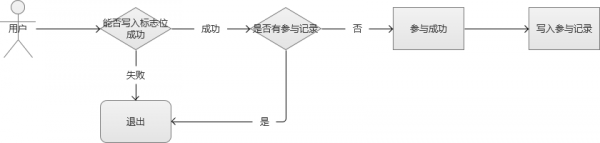
部分用户通过浏览器的插件或者其他工具，在秒杀开始的时间里，以自己的账号，一次发送上百甚至更多的请求。实际上，这样的用户破坏了秒杀和抢购的公平性。

这种请求在某些没有做数据安全处理的系统里，也可能造成另外一种破坏，导致某些判断条件被绕过。例如一个简单的领取逻辑，先判断用户是否有参与记录，如果没有则领取成功，最后写入到参与记录中。这是个非常简单的逻辑，但是，在高并发的场景下，存在深深的漏洞。多个并发请求通过负载均衡服务器，分配到内网的多台Web服务器，它们首先向存储发送查询请求，然后，在某个请求成功写入参与记录的时间差内，其他的请求获查询到的结果都是“没有参与记录”。这里，就存在逻辑判断被绕过的风险。



**应对方案：**

在程序入口处，一个账号只允许接受1个请求，其他请求过滤。不仅解决了同一个账号，发送N个请求的问题，还保证了后续的逻辑流程的安全。实现方案，可以通过Redis这种内存缓存服务，写入一个标志位（只允许1个请求写成功，结合watch的乐观锁的特性），成功写入的则可以继续参加。



或者，自己实现一个服务，将同一个账号的请求放入一个队列中，处理完一个，再处理下一个。

**2. 多个账号，一次性发送多个请求**

很多公司的账号注册功能，在发展早期几乎是没有限制的，很容易就可以注册很多个账号。因此，也导致了出现了一些特殊的工作室，通过编写自动注册脚本，积累了一大批“僵尸账号”，数量庞大，几万甚至几十万的账号不等，专门做各种刷的行为（这就是微博中的“僵尸粉“的来源）。举个例子，例如微博中有转发抽奖的活动，如果我们使用几万个“僵尸号”去混进去转发，这样就可以大大提升我们中奖的概率。

这种账号，使用在秒杀和抢购里，也是同一个道理。例如，iPhone官网的抢购，火车票黄牛党。



**应对方案：**

这种场景，可以通过检测指定机器IP请求频率就可以解决，如果发现某个IP请求频率很高，可以给它弹出一个验证码或者直接禁止它的请求：

1. 弹出验证码，最核心的追求，就是分辨出真实用户。因此，大家可能经常发现，网站弹出的验证码，有些是“鬼神乱舞”的样子，有时让我们根本无法看清。他们这样做的原因，其实也是为了让验证码的图片不被轻易识别，因为强大的“自动脚本”可以通过图片识别里面的字符，然后让脚本自动填写验证码。实际上，有一些非常创新的验证码，效果会比较好，例如给你一个简单问题让你回答，或者让你完成某些简单操作（例如百度贴吧的验证码）。
2. 直接禁止IP，实际上是有些粗暴的，因为有些真实用户的网络场景恰好是同一出口IP的，可能会有“误伤“。但是这一个做法简单高效，根据实际场景使用可以获得很好的效果。

**3. 多个账号，不同IP发送不同请求**

所谓道高一尺，魔高一丈。有进攻，就会有防守，永不休止。这些“工作室”，发现你对单机IP请求频率有控制之后，他们也针对这种场景，想出了他们的“新进攻方案”，就是不断改变IP。



有同学会好奇，这些随机IP服务怎么来的。有一些是某些机构自己占据一批独立IP，然后做成一个随机代理IP的服务，有偿提供给这些“工作室”使用。还有一些更为黑暗一点的，就是通过木马黑掉普通用户的电脑，这个木马也不破坏用户电脑的正常运作，只做一件事情，就是转发IP包，普通用户的电脑被变成了IP代理出口。通过这种做法，黑客就拿到了大量的独立IP，然后搭建为随机IP服务，就是为了挣钱。

**应对方案：**

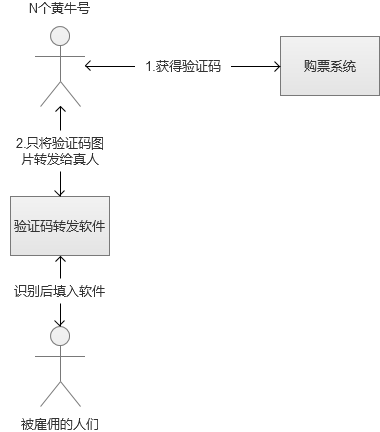
说实话，这种场景下的请求，和真实用户的行为，已经基本相同了，想做分辨很困难。再做进一步的限制很容易“误伤“真实用户，这个时候，通常只能通过设置业务门槛高来限制这种请求了，或者通过账号行为的”数据挖掘“来提前清理掉它们。

僵尸账号也还是有一些共同特征的，例如账号很可能属于同一个号码段甚至是连号的，活跃度不高，等级低，资料不全等等。根据这些特点，适当设置参与门槛，例如限制参与秒杀的账号等级。通过这些业务手段，也是可以过滤掉一些僵尸号。

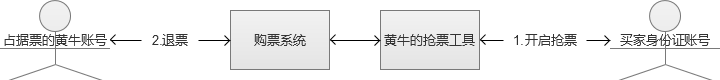
**4. 火车票的抢购**

看到这里，同学们是否明白你为什么抢不到火车票？如果你只是老老实实地去抢票，真的很难。通过多账号的方式，火车票的黄牛将很多车票的名额占据，部分强大的黄牛，在处理验证码方面，更是“技高一筹“。

高级的黄牛刷票时，在识别验证码的时候使用真实的人，中间搭建一个展示验证码图片的中转软件服务，真人浏览图片并填写下真实验证码，返回给中转软件。对于这种方式，验证码的保护限制作用被废除了，目前也没有很好的解决方案。



因为火车票是根据身份证实名制的，这里还有一个火车票的转让操作方式。大致的操作方式，是先用买家的身份证开启一个抢票工具，持续发送请求，黄牛账号选择退票，然后黄牛买家成功通过自己的身份证购票成功。当一列车厢没有票了的时候，是没有很多人盯着看的，况且黄牛们的抢票工具也很强大，即使让我们看见有退票，我们也不一定能抢得过他们哈。



最终，黄牛顺利将火车票转移到买家的身份证下。

**解决方案：**

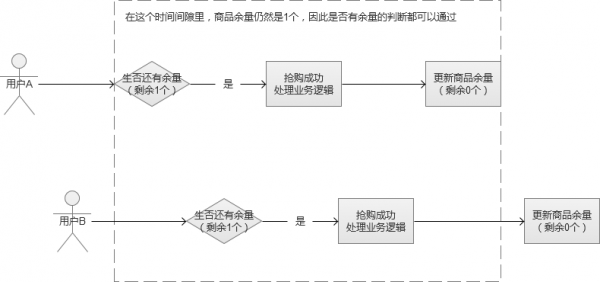
并没有很好的解决方案，唯一可以动心思的也许是对账号数据进行“数据挖掘”，这些黄牛账号也是有一些共同特征的，例如经常抢票和退票，节假日异常活跃等等。将它们分析出来，再做进一步处理和甄别。

**三、高并发下的数据安全**

我们知道在多线程写入同一个文件的时候，会存现“线程安全”的问题（多个线程同时运行同一段代码，如果每次运行结果和单线程运行的结果是一样的，结果和预期相同，就是线程安全的）。如果是MySQL数据库，可以使用它自带的锁机制很好的解决问题，但是，在大规模并发的场景中，是不推荐使用MySQL的。秒杀和抢购的场景中，还有另外一个问题，就是“超发”，如果在这方面控制不慎，会产生发送过多的情况。我们也曾经听说过，某些电商搞抢购活动，买家成功拍下后，商家却不承认订单有效，拒绝发货。这里的问题，也许并不一定是商家奸诈，而是系统技术层面存在超发风险导致的。

**1. 超发的原因**

假设某个抢购场景中，我们一共只有100个商品，在最后一刻，我们已经消耗了99个商品，仅剩最后一个。这个时候，系统发来多个并发请求，这批请求读取到的商品余量都是99个，然后都通过了这一个余量判断，最终导致超发。（同文章前面说的场景）

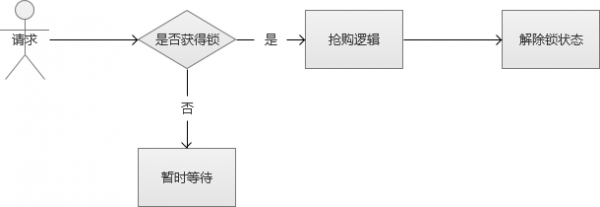


在上面的这个图中，就导致了并发用户B也“抢购成功”，多让一个人获得了商品。这种场景，在高并发的情况下非常容易出现。

**2. 悲观锁思路**

解决线程安全的思路很多，可以从“悲观锁”的方向开始讨论。

悲观锁，也就是在修改数据的时候，采用锁定状态，排斥外部请求的修改。遇到加锁的状态，就必须等待。



虽然上述的方案的确解决了线程安全的问题，但是，别忘记，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待“锁”，某些线程可能永远都没有机会抢到这个“锁”，这种请求就会死在那里。同时，这种请求会很多，瞬间增大系统的平均响应时间，结果是可用连接数被耗尽，系统陷入异常。

**3. FIFO队列思路**

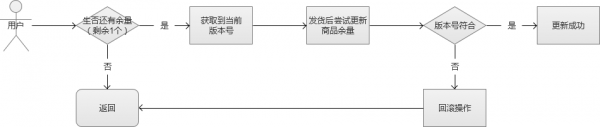
那好，那么我们稍微修改一下上面的场景，我们直接将请求放入队列中的，采用FIFO（First Input First Output，先进先出），这样的话，我们就不会导致某些请求永远获取不到锁。看到这里，是不是有点强行将多线程变成单线程的感觉哈。



然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬间将队列内存“撑爆”，然后系统又陷入到了异常状态。或者设计一个极大的内存队列，也是一种方案，但是，系统处理完一个队列内请求的速度根本无法和疯狂涌入队列中的数目相比。也就是说，队列内的请求会越积累越多，最终Web系统平均响应时候还是会大幅下降，系统还是陷入异常。

**4. 乐观锁思路**

这个时候，我们就可以讨论一下“乐观锁”的思路了。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。



有很多软件和服务都“乐观锁”功能的支持，例如Redis中的watch就是其中之一。通过这个实现，我们保证了数据的安全。

**5 ConcurrentHashMap**

是Java5中新增加的一个线程安全的Map集合，可以用来替代HashTable。HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

**四、小结**

互联网正在高速发展，使用互联网服务的用户越多，高并发的场景也变得越来越多。电商秒杀和抢购，是两个比较典型的互联网高并发场景。虽然我们解决问题的具体技术方案可能千差万别，但是遇到的挑战却是相似的，因此解决问题的思路也异曲同工。

# CAS原语无锁队列

# Docker是什么？

简单得来说，Docker是一个由GO语言写的程序运行的“容器”（Linux containers， LXCs）； 目前云服务的基石是操作系统级别的隔离，在同一台物理服务器上虚拟出多个主机。Docker则实现了一种应用程序级别的隔离； 它改变我们基本的开发、操作单元，由直接操作虚拟主机（VM）,转换到操作程序运行的“容器”上来。

Docker是为开发者和系统管理员设计的，用来发布和运行分布式应用程序的一个开放性平台。由两部分组成：

Docker Engine: 一个便携式、轻量级的运行环境和包管理器。（注\* 单OS vs 单线程，是不是跟NodeJS特别像？）

Docker Hub: 为创建自动化工作流和分享应用创建的云服务组成。（注\* 云端镜像/包管理 vs npm包管理，是不是跟npm特别像？）

从2013年3月20日，第一个版本的Docker正式发布到 2014年6月 [Docker 1.0 正式发布](http://blog.docker.com/2014/06/its-here-docker-1-0/)，经历了15个月。 虽然发展历程很短，但Docker正在有越来越流行的趋势。

其实Container技术并非Docker的创新，HeroKu, NodeJitsu 等云服务商都采用了类似这种轻量级的虚拟化技术，但Docker是第一个将这这种Container技术大规模开源并被社区广泛接受的。

好的部分

Docker相对于VM虚拟机的优势十分明显，那就是轻量和高性能和便捷性， 以下部分摘自： [KVM and Docker LXC Benchmarking with OpenStack](http://bodenr.blogspot.sg/2014/05/kvm-and-docker-lxc-benchmarking-with.html)

快

运行时的性能可以获取极大提升（经典的案例是提升97%）

管理操作（启动，停止，开始，重启等等） 都是以秒或毫秒为单位的。

敏捷

像虚拟机一样敏捷，而且会更便宜，在bare metal（裸机）上布署像点个按钮一样简单。

灵活

将应用和系统“容器化”，不添加额外的操作系统，

轻量

你会拥有足够的“操作系统”，仅需添加或减小镜像即可。在一台服务器上可以布署100~1000个Containers容器。

便宜

开源的，免费的，低成本的。由现代Linux内核支持并驱动。注\* 轻量的Container必定可以在一个物理机上开启更多“容器”，注定比VMs要便宜。

生态系统

正在越来越受欢迎，只需要看一看Google的趋势就知道了， [docker or LXC](http://www.google.com/trends/explore#q=/m/0wkcjgj&date=today 12-m&cmpt=q).

还有不计其数的社区和第三方应用。

云支持

不计其数的云服务提供创建和管理Linux容器框架。

有关Docker性能方面的优势，还可参考此IBM工程师对性能提升的评测，从各个方面比VMs（OS系统级别虚拟化）都有非常大的提升。

[Performance Characteristics of VMs vs Docker Containers by Boden Russel (IBM)](http://www.youtube.com/watch?v=JHqM_5X3MBU)

[Performance characteristics of traditional v ms vs docker containers](http://www.slideshare.net/dotCloud/performance-characteristics-of-traditional-v-ms-vs-docker-containers)

有争论的部分

任何项目都会有争论，就像Go，像NodeJS, 同样Docker也有一些。

能否彻底隔离

在超复杂的业务系统中，单OS到底能不能实现彻底隔离，一个程序的崩溃/内存溢出/高CPU占用到底会不会影响到其他容器或者整个系统？很多人对Docker能否在实际的多主机的生产环境中支持关键任务系统还有所怀疑。 注\* 就像有人质疑Node.JS单线程快而不稳，[无法在复杂场景中应用一样](http://ourjs.com/detail/%E6%8B%BF%E4%BB%80%E4%B9%88%E5%AE%88%E6%8A%A4%E4%BD%A0%E7%9A%84node-js%E8%BF%9B%E7%A8%8B-node%E5%87%BA%E9%94%99%E5%B4%A9%E6%BA%83%E4%BA%86%E6%80%8E%E4%B9%88%E5%8A%9E-)。

不过可喜的是，目前Linux内核已经针对Container做了很多改进，以支持更好的隔离。

GO语言还没有完全成熟

Docker由Go语言开发，但GO语言对大多数开发者来说比较陌生，而且还在不断改进，距离成熟还有一段时间。此半git、半包管理的方式让一些人产生不适。

被私有公司控制

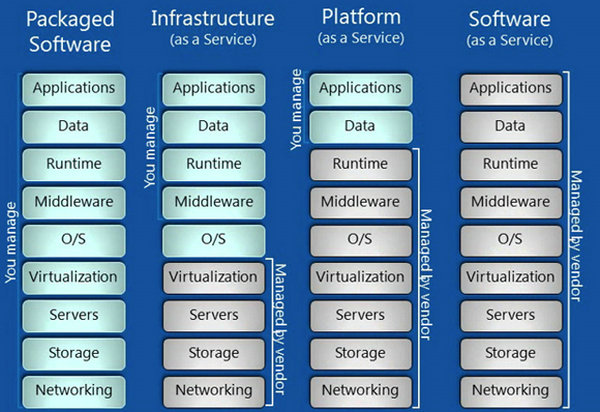
Docker是一家叫Dotcloud的私有公司设计的，公司都是以营利为目的，比如你没有办法使用源代码编绎Docker项目，只能使用黑匣子编出的Docker二进制发行包，未来可能不是完全免费的。 目前Docker已经推出面向公司的企业级服务（[咨询、支持和培训](http://blog.docker.com/2014/06/docker-announces-new-enterprise-support-training-and-consulting-services/)）。

# [云计算的三种服务模式：IaaS，PaaS和SaaS](http://www.cnblogs.com/beanmoon/archive/2012/12/10/2811547.html)

　　云服务”现在已经快成了一个家喻户晓的词了。如果你不知道PaaS, IaaS 和SaaS的区别，那么也没啥，因为很多人确实不知道。  
　　“云”其实是互联网的一个隐喻，“云计算”其实就是使用互联网来接入存储或者运行在远程服务器端的应用，数据，或者服务。  
　　任何一个使用基于互联网的方法来计算，存储和开发的公司，都可以从技术上叫做从事云的公司。然而，不是所有的云公司都一样。不是所有人都是CTO，所以有时候看到云技术背后的一些词可能会比较头疼。

**云也是分层的**

　　任何一个在互联网上提供其服务的公司都可以叫做云计算公司。其实云计算分几层的，分别是Infrastructure（基础设施）-as-a-Service，Platform（平台）-as-a-Service，Software（软件）-as-a-Service。基础设施在最下端，平台在中间，软件在顶端。别的一些“软”的层可以在这些层上面添加。



**IaaS: Infrastructure-as-a-Service（基础设施即服务）**

　　第一层叫做IaaS，有时候也叫做Hardware-as-a-Service，几年前如果你想在办公室或者公司的网站上运行一些企业应用，你需要去买服务器，或者别的高昂的硬件来控制本地应用，让你的业务运行起来。  
　　但是现在有IaaS，你可以将硬件外包到别的地方去。IaaS公司会提供场外服务器，存储和网络硬件，你可以租用。节省了维护成本和办公场地，公司可以在任何时候利用这些硬件来运行其应用。  
　　一些大的IaaS公司包括Amazon, Microsoft, VMWare, Rackspace和Red Hat.不过这些公司又都有自己的专长，比如Amazon和微软给你提供的不只是IaaS，他们还会将其计算能力出租给你来host你的网站。

**PaaS: Platform-as-a-Service（平台即服务）**

　　第二层就是所谓的PaaS，某些时候也叫做中间件。你公司所有的开发都可以在这一层进行，节省了时间和资源。  
　　PaaS公司在网上提供各种开发和分发应用的解决方案，比如虚拟服务器和操作系统。这节省了你在硬件上的费用，也让分散的工作室之间的合作变得更加容易。网页应用管理，应用设计，应用虚拟主机，存储，安全以及应用开发协作工具等。  
　　一些大的PaaS提供者有[Google App Engine](http://venturebeat.com/2011/11/14/cloud-iaas-paas-saas/),Microsoft Azure，Force.com,Heroku，[Engine Yard](http://venturebeat.com/2011/08/23/engine-yard-acquires-orchestra/)。最近兴起的公司有[AppFog](http://venturebeat.com/2011/08/11/appfog-raises-8m-to-host-powerful-web-apps-in-the-cloud/), [Mendix](http://venturebeat.com/2011/10/31/mendix-grabs-13m-to-fuel-fast-enterprise-app-development/) 和 [Standing Cloud](http://venturebeat.com/2011/11/10/standing-cloud-cloud-app-management/)

**SaaS: Software-as-a-Service（软件即服务）**

　　第三层也就是所谓SaaS。这一层是和你的生活每天接触的一层，大多是通过网页浏览器来接入。任何一个远程服务器上的应用都可以通过网络来运行，就是SaaS了。  
　　你消费的服务完全是从网页如Netflix, MOG, Google Apps, Box.net, Dropbox或者苹果的iCloud那里进入这些分类。尽管这些网页服务是用作商务和娱乐或者两者都有，但这也算是云技术的一部分。  
　　一些用作商务的SaaS应用包括Citrix的GoToMeeting，Cisco的WebEx，Salesforce的CRM，ADP，Workday和SuccessFactors。

**Iaas和Paas之间的比较**

    PaaS的主要作用是将一个开发和运行平台作为服务提供给用户，而IaaS的主要作用是提供虚拟机或者其他资源作为服务提供给用户。接下来，将在七个方面对PaaS和IaaS进行比较：

    1) 开发环境：PaaS基本都会给开发者提供一整套包括IDE在内的开发和测试环境，而IaaS方面用户主要还是沿用之前比较熟悉那套开发环境，但是因为之前那套开发环境在和云的整合方面比较欠缺，所以使用起来不是很方便。  
    2) 支持的应用：因为IaaS主要是提供虚拟机，而且普通的虚拟机能支持多种操作系统，所以IaaS支持的应用的范围是非常广泛的。但如果要让一个应用能跑在某个PaaS平台不是一件轻松的事，因为不仅需要确保这个应用是基于这个平台所支持的语言，而且也要确保这个应用只能调用这个平台所支持的API，如果这个应用调用了平台所不支持的API，那么就需要对这个应用进行修改。  
　3) 开放标准：虽然很多IaaS平台都存在一定的私有功能，但是由于OVF等协议的存在，使得IaaS在跨平台和避免被供应商锁定这两面是稳步前进的。而PaaS平台的情况则不容乐观，因为不论是Google的App Engine，还是Salesforce的Force.com都存在一定的私有API。  
    4) 可伸缩性：PaaS平台会自动调整资源来帮助运行于其上的应用更好地应对突发流量。而IaaS平台则需要开发人员手动对资源进行调整才能应对。  
    5) 整合率和经济性： PaaS平台整合率是非常高，比如PaaS的代表Google App Engine能在一台服务器上承载成千上万的应用，而普通的IaaS平台的整合率最多也不会超过100，而且普遍在10左右，使得IaaS的经济性不如PaaS。  
    6) 计费和监管：因为PaaS平台在计费和监管这两方面不仅达到了IaaS平台所能企及的操作系统层面，比如，CPU和内存的使用量等，而且还能做到应用层面，比如，应用的反应时间（Response Time）或者应用所消耗的事务多少等，这将提高计费和管理的精确性。  
    7) 学习难度：因为在IaaS上面开发和管理应用和现有的方式比较接近，而PaaS上面开发则有可能需要学一门新的语言或者新的框架，所以IaaS学习难度更低。

|  |  |  |
| --- | --- | --- |
|  | PaaS | IaaS |
| 开发环境 | 完善 | 普通 |
| 支持的应用 | 有限 | 广 |
| 通用性 | 欠缺 | 稍好 |
| 可伸缩性 | 自动伸缩 | 手动伸缩 |
| 整合率和经济性 | 高整合率，更经济 | 低整合率 |
| 计费和监管 | 精细 | 简单 |
| 学习难度 | 略难 | 低 |

表1. PaaS和IaaS之间的比较

未来的PK

    在当今云计算环境当中，IaaS是非常主流的，无论是Amazon EC2还是Linode或者Joyent等，都占有一席之地，但是随着Google的App Engine，Salesforce的Force.com还是微软的Windows Azure等PaaS平台的推出，使得PaaS也开始崭露头角。谈到这两者的未来，特别是这两者之间的竞争关系，我个人认为，短期而言，因为IaaS模式在支持的应用和学习难度这两方面的优势，使得IaaS将会在短期之内会成为开发者的首选，但是从长期而言，因为PaaS模式的高整合率所带来经济型使得如果PaaS能解决诸如通用性和支持的应用等方面的挑战，它将会替代IaaS成为开发者的“新宠”。

它们之间的关系主要可以从两个角度进行分析：其一是用户体验角度，从这个角度而言，它们之间关系是独立的，因为它们面对不同类型的用户。其二是技术角度，从这个角度而言，它们并不是简单的继承关系(Saa.....

**三种服务模式**

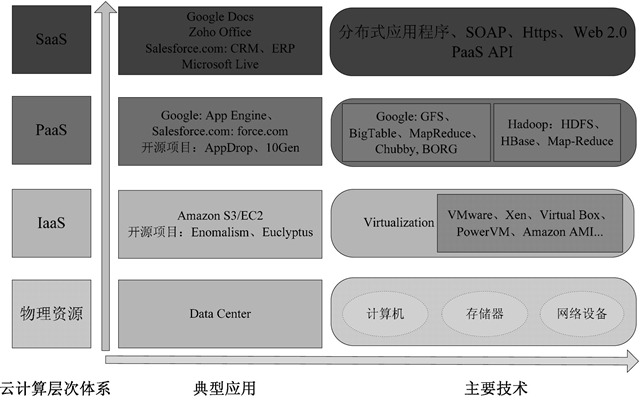
根据现在最常用，也是比较权威的NIST(National Institute of Standards and Technology，美国国家标准技术研究院)定义，云计算主要分为三种服务模式，而且这个三层的分法重要是从用户体验的角度出发的：

Software as a Service，软件即服务，简称SaaS，这层的作用是将应用作为服务提供给客户。

Platform as a Service，平台即服务，简称PaaS，这层的作用是将一个开发平台作为服务提供给用户。

Infrastructure as a Service， 基础设施即服务，简称IaaS，这层的作用是提供虚拟机或者其他资源作为服务提供给用户。

<注：此图不是原文配图>

[](http://images.cnblogs.com/cnblogs_com/afarmer/201112/201112051606143697.jpg)

**一、SaaS模式**

作用

通过SaaS这种模式，用户只要接上网络，并通过浏览器，就能直接使用在云端上运行的应用，而不需要顾虑类似安装等琐事，并且免去初期高昂的软硬件投入。SaaS主要面对的是普通的用户。

产品

主要产品包括：Salesforce Sales Cloud，Google Apps，Zimbra，Zoho和IBM Lotus Live等。

功能

谈到SaaS的功能，也可以认为是要实现SaaS服务，供应商需要完成那些功能?主要有四个方面：

随时随地访问：在任何时候或者任何地点，只要接上网络，用户就能访问这个SaaS服务。

支持公开协议：通过支持公开协议(比如HTML4/5)，能够方便用户使用。

安全保障：SaaS供应商需要提供一定的安全机制，不仅要使存储在云端的用户数据处于绝对安全的境地，而且也要在客户端实施一定的安全机制(比如HTTPS)来保护用户。

多住户(Multi-Tenant)机制：通过多住户机制，不仅能更经济地支撑庞大的用户规模，而且能提供一定的可定制性以满足用户的特殊需求。

**二、PaaS模式**

作用

通过PaaS这种模式，用户可以在一个包括SDK，文档和测试环境等在内的开发平台上非常方便地编写应用，而且不论是在部署，或者在运行的时候，用户都无需为服务器，操作系统，网络和存储等资源的管理操心，这些繁琐的工作都由PaaS供应商负责处理，而且PaaS在整合率上面非常惊人，比如一台运行Google App Engine的服务器能够支撑成千上万的应用，也就是说，PaaS是非常经济的。PaaS主要的用户是开发人员。

产品

主要产品包括：Google App Engine，force.com，heroku和Windows Azure Platform等。

功能

为了支撑着整个PaaS平台的运行，供应商需要提供那么功能?主要有四大功能：

友好的开发环境：通过提供SDK和IDE等工具来让用户能在本地方便地进行应用的开发和测试。

丰富的服务：PaaS平台会以API的形式将各种各样的服务提供给上层的应用。

自动的资源调度：也就是可伸缩这个特性，它将不仅能优化系统资源，而且能自动调整资源来帮助运行于其上的应用更好地应对突发流量。

精细的管理和监控：通过PaaS能够提供应用层的管理和监控，比如，能够观察应用运行的情况和具体数值(比如，吞吐量和反映时间)来更好地衡量应用的运行状态，还有能够通过精确计量应用使用所消耗的资源来更好地计费。

**三、IaaS模式**

作用

通过IaaS这种模式，用户可以从供应商那里获得他所需要的虚拟机或者存储等资源来装载相关的应用，同时这些基础设施的繁琐的管理工作将由IaaS供应商来处理。IaaS能通过它上面对虚拟机支持众多的应用。IaaS主要的用户是系统管理员。

产品

主要产品包括：Amazon EC2，Linode，Joyent，Rackspace，IBM Blue Cloud和Cisco UCS等。

功能

IaaS供应商需要在那些方面对基础设施进行管理以给用户提供资源?或者说IaaS云有那些功能?在《虚拟化与云计算》中列出了IaaS的七个基本功能：

资源抽象：使用资源抽象的方法(比如，资源池)能更好地调度和管理物理资源。

资源监控：通过对资源的监控，能够保证基础实施高效率的运行。

负载管理：通过负载管理，不仅能使部署在基础设施上的应用运能更好地应对突发情况，而且还能更好地利用系统资源。

数据管理：对云计算而言，数据的完整性，可靠性和可管理性是对IaaS的基本要求。

资源部署：也就是将整个资源从创建到使用的流程自动化。

安全管理：IaaS的安全管理的主要目标是保证基础设施和其提供的资源能被合法地访问和使用。

计费管理：通过细致的计费管理能使用户更灵活地使用资源。

接下来，稍微给大家介绍一下云的三种形式和云计算好处。

**三种模式之间的关系**

它们之间的关系主要可以从两个角度进行分析：其一是用户体验角度，从这个角度而言，它们之间关系是独立的，因为它们面对不同类型的用户。其二是技术角度，从这个角度而言，它们并不是简单的继承关系(SaaS基于PaaS，而PaaS基于IaaS)，因为首先SaaS可以是基于PaaS或者直接部署于IaaS之上，其次PaaS可以构建于IaaS之上，也可以直接构建在物理资源之上。

# Hadoop核心架构HDFS+MapReduce+Hbase+Hive内部机理详解

发表于2014-02-17 11:55| 48277次阅读| 来源CSDN博客| 10 条评论| 作者张震

**摘要：**通过对Hadoop分布式计算平台最核心的分布式文件系统HDFS、MapReduce处理过程，以及数据仓库工具Hive和分布式数据库Hbase的介绍，基本涵盖了Hadoop分布式平台的所有技术核心。

编者按：HDFS和MapReduce是Hadoop的两大核心，除此之外Hbase、Hive这两个核心工具也随着Hadoop发展变得越来越重要。本文作者[张震](http://blog.csdn.net/yczws1/article/details/19178265#1536434-tsina-1-98976-66a1f5d8f89e9ad52626f6f40fdeadaa)的博文《[Thinking in BigDate（八）大数据Hadoop核心架构HDFS+MapReduce+Hbase+Hive内部机理详解](http://blog.csdn.net/yczws1/article/details/19178265#1536434-tsina-1-98976-66a1f5d8f89e9ad52626f6f40fdeadaa)》从内部机理详细的分析了HDFS、MapReduce、Hbase、Hive的运行机制，从底层到数据管理详细的将Hadoop进行了一个剖析。

**CSDN推荐：欢迎免费订阅《**[**Hadoop与大数据周刊**](http://g.csdn.net/5261396)**》获取更多Hadoop技术文献、大数据技术分析、企业实战经验，生态圈发展趋势。**

**以下是作者原文：**

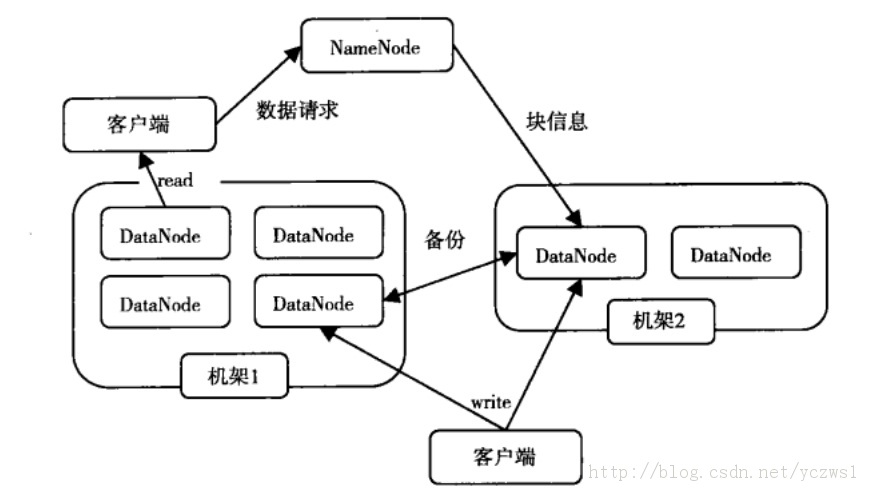
通过这一阶段的调研总结，从内部机理的角度详细分析，HDFS、MapReduce、Hbase、Hive是如何运行，以及基于Hadoop数据仓库的构建和分布式数据库内部具体实现。如有不足，后续及时修改。

**HDFS的体系架构**

整个Hadoop的体系结构主要是通过HDFS来实现对分布式存储的底层支持，并通过MR来实现对分布式并行任务处理的程序支持。

HDFS采用主从（Master/Slave）结构模型，一个HDFS集群是由一个NameNode和若干个DataNode组成的（在最新的Hadoop2.2版本已经实现多个NameNode的配置-这也是一些大公司通过修改hadoop源代码实现的功能，在最新的版本中就已经实现了）。NameNode作为主服务器，管理文件系统命名空间和客户端对文件的访问操作。DataNode管理存储的数据。HDFS支持文件形式的数据。

从内部来看，文件被分成若干个数据块，这若干个数据块存放在一组DataNode上。NameNode执行文件系统的命名空间，如打开、关闭、重命名文件或目录等，也负责数据块到具体DataNode的映射。DataNode负责处理文件系统客户端的文件读写，并在NameNode的统一调度下进行数据库的创建、删除和复制工作。NameNode是所有HDFS元数据的管理者，用户数据永远不会经过NameNode。

[](http://cms.csdnimg.cn/article/201402/17/530186b9549fa.jpg)

**如图：HDFS体系结构图**

图中涉及三个角色：NameNode、DataNode、Client。NameNode是管理者，DataNode是文件存储者、Client是需要获取分布式文件系统的应用程序。

**文件写入：**

   1）  Client向NameNode发起文件写入的请求。

   2）  NameNode根据文件大小和文件块配置情况，返回给Client它管理的DataNode的信息。

   3）  Client将文件划分为多个block，根据DataNode的地址，按顺序将block写入DataNode块中。

**文件读取：**

   1）  Client向NameNode发起读取文件的请求。

   2）  NameNode返回文件存储的DataNode信息。

   3）  Client读取文件信息。

      HDFS作为分布式文件系统在数据管理方面可借鉴点：

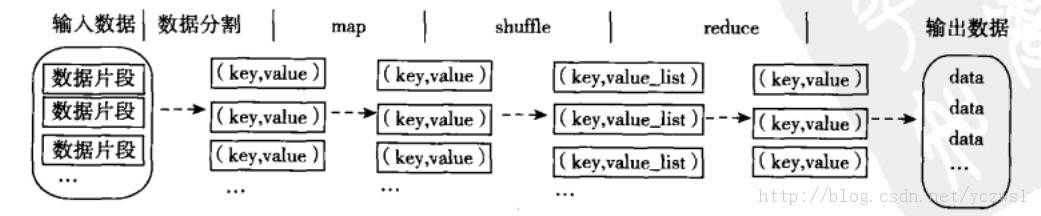
      文件块的放置：一个Block会有三份备份，一份在NameNode指定的DateNode上，一份放在与指定的DataNode不在同一台机器的DataNode上，一根在于指定的DataNode在同一Rack上的DataNode上。备份的目的是为了数据安全，采用这种方式是为了考虑到同一Rack失败的情况，以及不同数据拷贝带来的性能的问题。

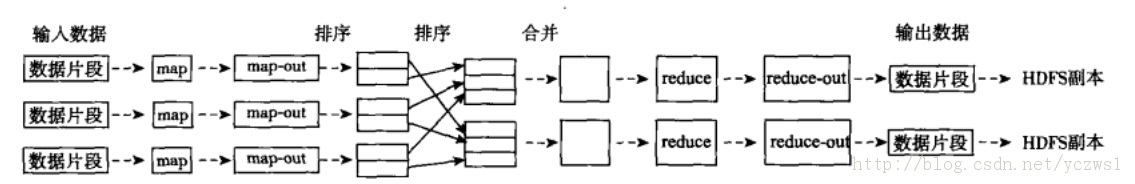
**MapReduce体系架构**

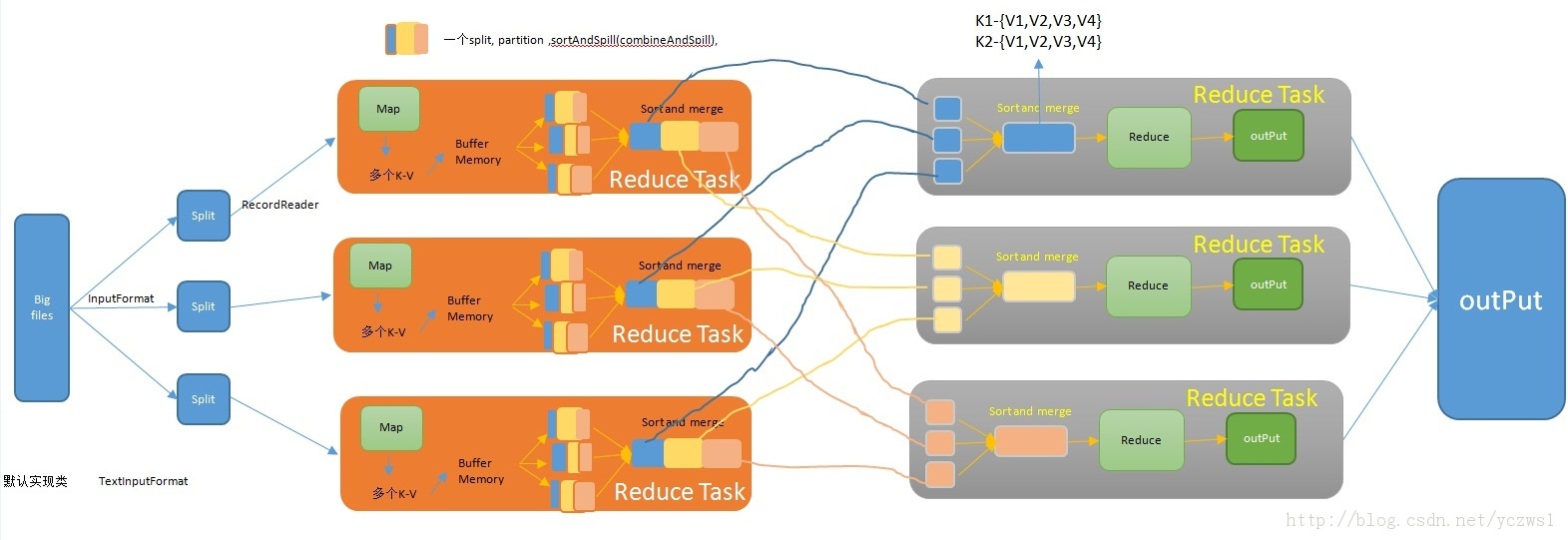
      MR框架是由一个单独运行在主节点上的JobTracker和运行在每个集群从节点上的TaskTracker共同组成。主节点负责调度构成一个作业的所有任务，这些任务分布在不同的不同的从节点上。主节点监视它们的执行情况，并重新执行之前失败的任务。从节点仅负责由主节点指派的任务。当一个Job被提交时，JobTracker接受到提交作业和配置信息之后，就会将配置信息等分发给从节点，同时调度任务并监控TaskTracker的执行。JobTracker可以运行于集群中的任意一台计算机上。TaskTracker负责执行任务，它必须运行在DataNode上，DataNode既是数据存储节点，也是计算节点。JobTracker将map任务和reduce任务分发给空闲的TaskTracker，这些任务并行运行，并监控任务运行的情况。如果JobTracker出了故障，JobTracker会把任务转交给另一个空闲的TaskTracker重新运行。

       HDFS和MR共同组成Hadoop分布式系统体系结构的核心。**HDFS在集群上实现了分布式文件系统，MR在集群上实现了分布式计算和任务处理。**HDFS在MR任务处理过程中提供了文件操作和存储等支持，MR在HDFS的基础上实现了任务的分发、跟踪、执行等工作，并收集结果，二者相互作用，完成分布式集群的主要任务。

        Hadoop上的并行应用程序开发是基于MR编程框架。MR编程模型原理：利用一个输入的key-value对集合来产生一个输出的key-value对集合。MR库通过Map和Reduce两个函数来实现这个框架。用户自定义的map函数接受一个输入的key-value对，然后产生一个中间的key-value对的集合。**MR把所有具有相同的key值的value结合在一起，然后传递个reduce函数。**Reduce函数接受key和相关的value结合，reduce函数合并这些value值，形成一个较小的value集合。通常我们通过一个迭代器把中间的value值提供给reduce函数（迭代器的作用就是收集这些value值），这样就可以处理无法全部放在内存中的大量的value值集合了。

[](http://cms.csdnimg.cn/article/201402/17/530187136fb9d.jpg)

[](http://cms.csdnimg.cn/article/201402/17/53018719001af.jpg)

[](http://cms.csdnimg.cn/article/201402/17/5301871d98261.jpg)

       说明：（第三幅图为同伴自己画的）

       流程简而言之，大数据集被分成众多小的数据集块，若干个数据集被分在集群中的一个节点进行处理并产生中间结果。单节点上的任务，map函数一行行读取数据获得数据的（k1,v1），数据进入缓存，通过map函数执行map（基于key-value）排序（框架会对map的输出进行排序）执行后输入（k2,v2）。每一台机器都执行同样的操作。不同机器上的（k2,v2）通过merge排序的过程（shuffle的过程可以理解成reduce前的一个过程），最后reduce合并得到，（k3,v3），输出到HDFS文件中。

       谈到reduce，在reduce之前，可以先对中间数据进行数据合并（Combine），即将中间有相同的key的<key,value>对合并。Combine的过程与reduce的过程类似，但Combine是作为map任务的一部分，在执行完map函数后仅接着执行。Combine能减少中间结果key-value对的数目，从而降低网络流量。

       Map任务的中间结果在做完Combine和Partition后，以文件的形式存于本地磁盘上。中间结果文件的位置会通知主控JobTracker，JobTracker再通知reduce任务到哪一个DataNode上去取中间结果。所有的map任务产生的中间结果均按其key值按hash函数划分成R份，R个reduce任务各自负责一段key区间。每个reduce需要向许多个map任务节点取的落在其负责的key区间内的中间结果，然后执行reduce函数，最后形成一个最终结果。有R个reduce任务，就会有R个最终结果，很多情况下这R个最终结果并不需要合并成一个最终结果，因为这R个最终结果可以作为另一个计算任务的输入，开始另一个并行计算任务。这就形成了上面图中多个输出数据片段（HDFS副本）。

**Hbase数据管理**

      Hbase就是Hadoop database。与传统的mysql、oracle究竟有什么差别。即列式数据与行式数据由什么区别。NoSql数据库与传统关系型数据由什么区别：

**Hbase VS Oracle**

     1、  Hbase适合大量插入同时又有读的情况。输入一个Key获取一个value或输入一些key获得一些value。

     2、  Hbase的瓶颈是硬盘传输速度。Hbase的操作，它可以往数据里面insert，也可以update一些数据，但update的实际上也是insert，只是插入一个新的时间戳的一行。Delete数据，也是insert，只是insert一行带有delete标记的一行。Hbase的所有操作都是追加插入操作。Hbase是一种日志集数据库。它的存储方式，像是日志文件一样。它是批量大量的往硬盘中写，通常都是以文件形式的读写。这个读写速度，就取决于硬盘与机器之间的传输有多快。而Oracle的瓶颈是硬盘寻道时间。它经常的操作时随机读写。要update一个数据，先要在硬盘中找到这个block，然后把它读入内存，在内存中的缓存中修改，过段时间再回写回去。由于你寻找的block不同，这就存在一个随机的读。硬盘的寻道时间主要由转速来决定的。而寻道时间，技术基本没有改变，这就形成了寻道时间瓶颈。

     3、  Hbase中数据可以保存许多不同时间戳的版本（即同一数据可以复制许多不同的版本，准许数据冗余，也是优势）。数据按时间排序，因此Hbase特别适合寻找按照时间排序寻找Top n的场景。找出某个人最近浏览的消息，最近写的N篇博客，N种行为等等，因此Hbase在互联网应用非常多。

     4、  Hbase的局限。只能做很简单的Key-value查询。它适合有高速插入，同时又有大量读的操作场景。而这种场景又很极端，并不是每一个公司都有这种需求。在一些公司，就是普通的OLTP（联机事务处理）随机读写。在这种情况下，Oracle的可靠性，系统的负责程度又比Hbase低一些。而且Hbase局限还在于它只有主键索引，因此在建模的时候就遇到了问题。比如，在一张表中，很多的列我都想做某种条件的查询。但却只能在主键上建快速查询。所以说，不能笼统的说那种技术有优势。

     5、 Oracle是行式数据库，而Hbase是列式数据库。列式数据库的优势在于数据分析这种场景。数据分析与传统的OLTP的区别。数据分析，经常是以某个列作为查询条件，返回的结果也经常是某一些列，不是全部的列。在这种情况下，行式数据库反应的性能就很低效。

      行式数据库：Oracle为例，数据文件的基本组成单位：块/页。块中数据是按照一行行写入的。这就存在一个问题，当我们要读一个块中的某些列的时候，不能只读这些列，必须把这个块整个的读入内存中，再把这些列的内容读出来。换句话就是：为了读表中的某些列，必须要把整个表的行全部读完，才能读到这些列。这就是行数据库最糟糕的地方。

      列式数据库：是以列作为元素存储的。同一个列的元素会挤在一个块。当要读某些列，只需要把相关的列块读到内存中，这样读的IO量就会少很多。通常，同一个列的数据元素通常格式都是相近的。这就意味着，当数据格式相近的时候，数据就可以做大幅度的压缩。所以，列式数据库在数据压缩方面有很大的优势，压缩不仅节省了存储空间，同时也节省了IO。（这一点，可利用在当数据达到百万、千万级别以后，数据查询之间的优化，提高性能，示场景而定）

**Hive数据管理**

       Hive是建立在Hadoop上的数据仓库基础架构。它提供了一系列的工具，用来进行数据提取、转换、加载，这是一种可以存储、查询和分析存储在Hadoop中的大规模数据机制。可以把Hadoop下结构化数据文件映射为一张成Hive中的表，并提供类sql查询功能，除了不支持更新、索引和事务，sql其它功能都支持。可以将sql语句转换为MapReduce任务进行运行，作为sql到MapReduce的映射器。提供shell、JDBC/ODBC、Thrift、Web等接口。优点：成本低可以通过类sql语句快速实现简单的MapReduce统计。作为一个数据仓库，Hive的数据管理按照使用层次可以从元数据存储、数据存储和数据交换三个方面介绍。

**（1）元数据存储**

     Hive将元数据存储在RDBMS中，有三种方式可以连接到数据库：

     ·内嵌模式：元数据保持在内嵌数据库的Derby，一般用于单元测试，只允许一个会话连接

     ·多用户模式：在本地安装Mysql，把元数据放到Mysql内

     ·远程模式：元数据放置在远程的Mysql数据库

**（2）数据存储**

      首先，Hive没有专门的数据存储格式，也没有为数据建立索引，用于可以非常自由的组织Hive中的表，只需要在创建表的时候告诉Hive数据中的列分隔符和行分隔符，这就可以解析数据了。

      其次，Hive中所有的数据都存储在HDFS中，Hive中包含4中数据模型：Tabel、ExternalTable、Partition、Bucket。

      Table：类似与传统数据库中的Table，每一个Table在Hive中都有一个相应的目录来存储数据。例如：一个表zz，它在HDFS中的路径为：/wh/zz，其中wh是在hive-site.xml中由${hive.metastore.warehouse.dir}指定的数据仓库的目录，所有的Table数据（不含External Table）都保存在这个目录中。

      Partition：类似于传统数据库中划分列的索引。在Hive中，表中的一个Partition对应于表下的一个目录，所有的Partition数据都存储在对应的目录中。例如：zz表中包含ds和city两个Partition，则对应于ds=20140214，city=beijing的HDFS子目录为：/wh/zz/ds=20140214/city=Beijing;

      Buckets：对指定列计算的hash，根据hash值切分数据，目的是为了便于并行，每一个Buckets对应一个文件。将user列分数至32个Bucket上，首先对user列的值计算hash，比如，对应hash=0的HDFS目录为：/wh/zz/ds=20140214/city=Beijing/part-00000;对应hash=20的，目录为：/wh/zz/ds=20140214/city=Beijing/part-00020。

      ExternalTable指向已存在HDFS中的数据，可创建Partition。和Table在元数据组织结构相同，在实际存储上有较大差异。Table创建和数据加载过程，可以用统一语句实现，实际数据被转移到数据仓库目录中，之后对数据的访问将会直接在数据仓库的目录中完成。删除表时，表中的数据和元数据都会删除。ExternalTable只有一个过程，因为加载数据和创建表是同时完成。世界数据是存储在Location后面指定的HDFS路径中的，并不会移动到数据仓库中。

**（3）数据交换**

     ·用户接口：包括客户端、Web界面和数据库接口

     ·元数据存储：通常是存储在关系数据库中的，如Mysql，Derby等

     ·Hadoop：用HDFS进行存储，利用MapReduce进行计算。

      关键点：Hive将元数据存储在数据库中，如Mysql、Derby中。Hive中的元数据包括表的名字、表的列和分区及其属性、表的属性（是否为外部表）、表数据所在的目录等。

      Hive的数据存储在HDFS中，大部分的查询由MapReduce完成。

**总结：**

        通过对Hadoop分布式计算平台最核心的分布式文件系统HDFS、MapReduce处理过程，以及数据仓库工具Hive和分布式数据库Hbase的介绍。基本涵盖了Hadoop分布式平台的所有技术核心。从体系架构到数据定义到数据存储再到数据处理，从宏观到微观的系统介绍，为Hadoop平台上大规模的数据存储和任务处理打下基础。

# [Spark使用总结与分享](http://www.cnblogs.com/bourneli/p/4394271.html)

**背景**

使用spark开发已有几个月。相比于python/hive，scala/spark学习门槛较高。尤其记得刚开时，举步维艰，进展十分缓慢。不过谢天谢地，这段苦涩（bi）的日子过去了。忆苦思甜，为了避免项目组的其他同学走弯路，决定总结和梳理spark的使用经验。

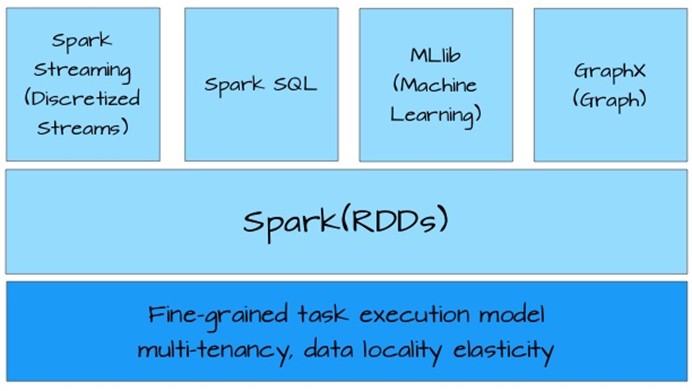
**Spark基础**

基石RDD

spark的核心是RDD（弹性分布式数据集），一种通用的数据抽象，封装了基础的数据操作，如map，filter，reduce等。RDD提供**数据共享**的抽象，相比其他大数据处理框架，如MapReduce，Pegel，DryadLINQ和HIVE等均缺乏此特性，所以RDD更为通用。

简要地概括RDD：RDD是一个不可修改的，分布的对象集合。每个RDD由多个分区组成，每个分区可以同时在集群中的不同节点上计算。RDD可以包含Python，Java和Scala中的任意对象。

Spark生态圈中应用都是基于RDD构建（下图），这一点充分说明RDD的抽象足够通用，可以描述大多数应用场景。

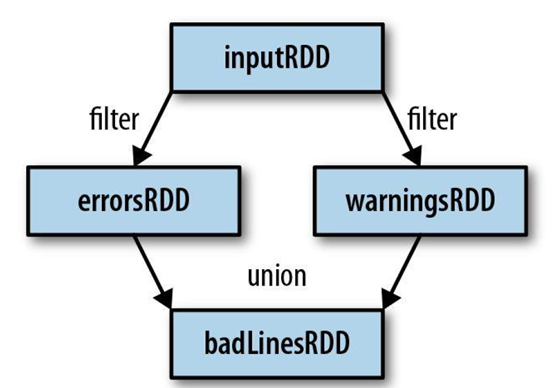


RDD操作类型—转换和动作

RDD的操作主要分两类：转换（transformation）和动作（action）。两类函数的主要区别是，转换接受RDD并返回RDD，而动作接受RDD但是返回非RDD。转换采用惰性调用机制，每个RDD记录父RDD转换的方法，这种调用链表称之为血缘（lineage）；而动作调用会直接计算。

采用惰性调用，通过血缘连接的RDD操作可以管道化（pipeline），管道化的操作可以直接在单节点完成，避免**多次转换操作之间数据同步的等待**。

使用血缘串联的操作可以保持每次计算相对简单，而不用担心有过多的中间数据，因为这些血缘操作都管道化了，这样也保证了逻辑的单一性，而不用像MapReduce那样，为了竟可能的减少map reduce过程，在单个map reduce中写入过多复杂的逻辑。



RDD使用模式

RDD使用具有一般的模式，可以抽象为下面的几步

加载外部数据，创建RDD对象

使用转换（如filter），创建新的RDD对象

缓存需要重用的RDD

使用动作（如count），启动并行计算

RDD高效的策略

Spark官方提供的数据是RDD在某些场景下，计算效率是Hadoop的20X。这个数据是否有水分，我们先不追究，但是RDD效率高的由一定机制保证的：

RDD数据只读，不可修改。如果需要修改数据，必须从父RDD转换（transformation）到子RDD。所以，在容错策略中，**RDD没有数据冗余**，而是通过RDD父子依赖（血缘）关系进行重算实现容错。

RDD数据在内存中，多个RDD操作之间，数据不用落地到磁盘上，避免不必要的I/O操作。

RDD存放的数据可以是java对象，所以避免的不必要的对象序列化和反序列化。

总而言之，RDD高效的主要因素是尽量避免不必要的操作和牺牲数据的操作精度，用来提高计算效率。

**Spark使用技巧**

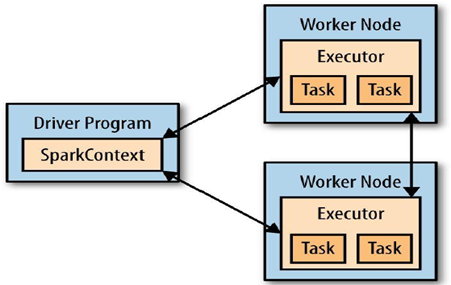
RDD基本函数扩展

RDD虽然提供了很多函数，但是毕竟还是有限的，有时候需要扩展，自定义新的RDD的函数。在spark中，可以通过隐式转换，轻松实现对RDD扩展。画像开发过程中，平凡的会使用rollup操作（类似HIVE中的rollup），计算多个级别的聚合数据。下面是具体实，

|  |
| --- |
| /\*\*  \* 扩展spark rdd,为rdd提供rollup方法  \*/  implicit class RollupRDD[T: ClassTag](rdd: RDD[(Array[String], T)]) extends Serializable {    /\*\*  \* 类似Sql中的rollup操作  \*  \* @param aggregate 聚合函数  \* @param keyPlaceHold key占位符，默认采用FaceConf.STAT\_SUMMARY  \* @param isCache，确认是否缓存数据  \* @return 返回聚合后的数据  \*/  def rollup[U: ClassTag](  aggregate: Iterable[T] => U,  keyPlaceHold: String = FaceConf.STAT\_SUMMARY,  isCache: Boolean = true): RDD[(Array[String], U)] = {    if (rdd.take(1).isEmpty) {  return rdd.map(x => (Array[String](), aggregate(Array[T](x.\_2))))  }    if (isCache) {  rdd.cache // 提高计算效率  }  val totalKeyCount = rdd.first.\_1.size  val result = { 1 to totalKeyCount }.par.map(untilKeyIndex => { // 并行计算  rdd.map(row => {  val combineKey = row.\_1.slice(0, untilKeyIndex).mkString(FaceConf.KEY\_SEP) // 组合key  (combineKey, row.\_2)  }).groupByKey.map(row => { // 聚合计算  val oldKeyList = row.\_1.split(FaceConf.KEY\_SEP)  val newKeyList = oldKeyList ++ Array.fill(totalKeyCount - oldKeyList.size) { keyPlaceHold }  (newKeyList, aggregate(row.\_2))  })  }).reduce(\_ ++ \_) // 聚合结果    result  }    } |

上面代码声明了一个隐式类，具有一个成员变量rdd，类型是RDD[(Array[String], T)]，那么如果应用代码中出现了任何这样的rdd对象，并且import当前的隐式转换，那么编译器就会将这个rdd当做上面的隐式类的对象，也就可以使用rollup函数，和一般的map，filter方法一样。

RDD操作闭包外部变量原则



RDD相关操作都需要传入自定义闭包函数（closure），如果这个函数需要访问外部变量，那么需要遵循一定的规则，否则会抛出运行时异常。闭包函数传入到节点时，需要经过下面的步骤：

1. 驱动程序，通过反射，运行时找到闭包访问的所有变量，并封成一个对象，然后序列化该对象
2. 将序列化后的对象通过网络传输到worker节点
3. worker节点反序列化闭包对象
4. worker节点执行闭包函数，

注意：外部变量在闭包内的修改**不会被反馈**到驱动程序。

简而言之，就是通过网络，传递函数，然后执行。所以，被传递的变量必须可以**序列化**，否则传递失败。本地执行时，仍然会执行上面四步。

广播机制也可以做到这一点，但是频繁的使用广播会使代码不够简洁，而且广播设计的初衷是将较大数据缓存到节点上，避免多次数据传输，提高计算效率，而不是用于进行外部变量访问。

RDD数据同步

RDD目前提供两个数据同步的方法：广播和累计器。

**广播 broadcast**

前面提到过，广播可以将变量发送到闭包中，被闭包使用。但是，广播还有一个作用是同步较大数据。比如你有一个IP库，可能有几G，在map操作中，依赖这个ip库。那么，可以通过广播将这个ip库传到闭包中，被并行的任务应用。广播通过两个方面提高数据共享效率：1，集群中每个节点（物理机器）只有一个副本，默认的闭包是每个任务一个副本；2，广播传输是通过BT下载模式实现的，也就是P2P下载，在集群多的情况下，可以极大的提高数据传输速率。广播变量修改后，不会反馈到其他节点。

**累加器 Accumulator**

累加器是一个write-only的变量，用于累加各个任务中的状态，只有在驱动程序中，才能访问累加器。而且，截止到1.2版本，累加器有一个已知的缺陷，在action操作中，n个元素的RDD可以确保累加器只累加n次，但是在transformation时，spark不确保，也就是累加器可能出现n+1次累加。

目前RDD提供的同步机制粒度太粗，尤其是转换操作中变量状态不能同步，所以RDD无法做复杂的具有状态的事务操作。不过，RDD的使命是提供一个通用的并行计算框架，估计永远也不会提供细粒度的数据同步机制，因为这与其设计的初衷是违背的。

RDD优化技巧

**RDD缓存**

需要使用多次的数据需要cache，否则会进行不必要的重复操作。举个例子

|  |
| --- |
| val data = … // read from tdw  println(data.filter(\_.contains("error")).count)  println(data.filter(\_.contains("warning")).count) |

上面三段代码中，data变量会加载两次，高效的做法是在data加载完后，立刻持久化到内存中，如下

|  |
| --- |
| val data = … // read from tdw  data.cache  println(data.filter(\_.contains("error")).count)  println(data.filter(\_.contains("warning")).count) |

这样，data在第一加载后，就被缓存到内存中，后面两次操作均直接使用内存中的数据。

**转换并行化**

RDD的转换操作时并行化计算的，但是多个RDD的转换同样是可以并行的，参考如下

|  |
| --- |
| val dataList:Array[RDD[Int]] = …  val sumList = data.list.map(\_.map(\_.sum)) |

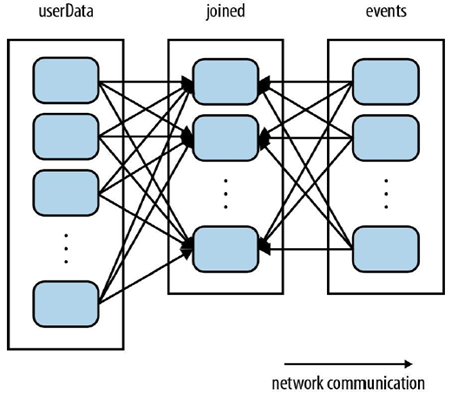
上面的例子中，第一个map是便利Array变量，串行的计算每个RDD中的每行的sum。由于每个RDD之间计算是没有逻辑联系的，所以理论上是可以将RDD的计算并行化的，在scala中可以轻松试下，如下

|  |
| --- |
| val dataList:Array[RDD[Int]] = …  val sumList = data.list.par.map(\_.map(\_.sum)) |

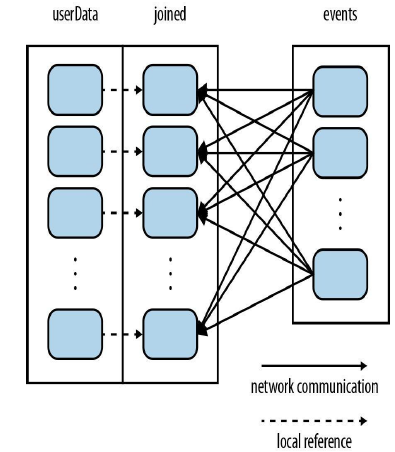
注意红色代码。

**减少shuffle网络传输**

一般而言，网络I/O开销是很大的，减少网络开销，可以显著加快计算效率。任意两个RDD的shuffle操作（join等）的大致过程如下，



用户数据userData和事件events数据通过用户id连接，那么会在网络中传到另外一个节点，这个过程中，有两个网络传输过程。Spark的默认是完成这两个过程。但是，如果你多告诉spark一些信息，spark可以优化，只执行一个网络传输。可以通过使用、HashPartition，在userData"本地"先分区，然后要求events直接shuffle到userData的节点上，那么就减少了一部分网络传输，减少后的效果如下，



虚线部分都是在本地完成的，没有网络传输。在数据加载时，就按照key进行partition，这样可以经一部的减少本地的HashPartition的过程，示例代码如下

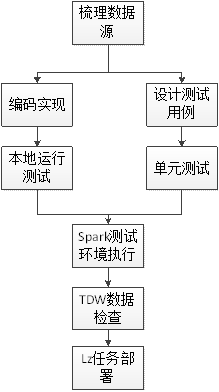
|  |
| --- |
| val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://…")  .partitionBy(new HashPartitioner(100)) // Create 100 partitions  .persist() |

注意，上面一定要persist，否则会重复计算多次。100用来指定并行数量。

**Spark其他**

Spark开发模式

由于spark应用程序是需要在部署到集群上运行的，导致本地调试比较麻烦，所以经过这段时间的经验累积，总结了一套开发流程，目的是为了尽可能的提高开发调试效率，同时保证开发质量。当然，这套流程可能也不是最优的，后面需要持续改进。



整个流程比较清楚，这里主要谈谈为什么需要单元测试。公司内的大多数项目，一般不提倡单元测试，而且由于项目进度压力，开发人员会非常抵触单元测试，因为会花费"额外"的精力。Bug这东西不会因为项目赶进度而消失，而且恰好相反，可能因为赶进度，而高于平均水平。所以，如果不花时间进行单元测试，那么会花同样多，甚至更多的时间调试。很多时候，往往一些很小的bug，却导致你花了很长时间去调试，而这些bug，恰好是很容易在单元测试中发现的。而且，单元测试还可以带来两个额外的好处：1）API使用范例；2）回归测试。所以，还是单元测试吧，这是一笔投资，而且ROI还挺高！不过凡事需要掌握分寸，单元测试应该根据项目紧迫程度调整粒度，做到有所为，有所不为。

Spark其他功能

前面提到了spark生态圈，spark除了核心的RDD，还提供了之上的几个很使用的应用：

Spark SQL: 类似hive，使用rdd实现sql查询

Spark Streaming: 流式计算，提供实时计算功能，类似storm

MLLib：机器学习库，提供常用分类，聚类，回归，交叉检验等机器学习算法并行实现。

GraphX：图计算框架，实现了基本的图计算功能，常用图算法和pregel图编程框架

# [ELK学习总结（1-1）ELK是什么](http://www.cnblogs.com/lexiaofei/p/6548528.html)

**1、elk 是什么 ？**

     Elastic Stack（旧称ELK Stack），是一种能够从任意数据源抽取数据，并实时对数据进行搜索、分析和可视化展现的数据分析框架。（hadoop同一个开发人员）

     java 开发的开源的全文搜索引擎工具

     基于lucence搜索引擎的

     采用 restful - api 标准的

     高可用、高扩展的分布式框架

     实时数据分析的

     官方网站： <https://www.elastic.co/products>

**2、为什么要用elk？**

     服务器众多，组件众多，日志众多

     发现问题困难，技能要求高

**业务场景：《实时日志分析展现》**

     日志主要包括系统日志、应用程序日志和安全日志。

     系统运维和开发人员可以通过日志了解服务器软硬件信息、检查配置过程中的错误及错误发生的原因。经常分析日志可以了解服务器的负荷，性能安全性，从而及时采取措施纠正错误。

     通常，日志被分散的储存不同的设备上。如果你管理数十上百台服务器，你还在使用依次登录每台机器的传统方法查阅日志。这样是不是感觉很繁琐和效率低下。

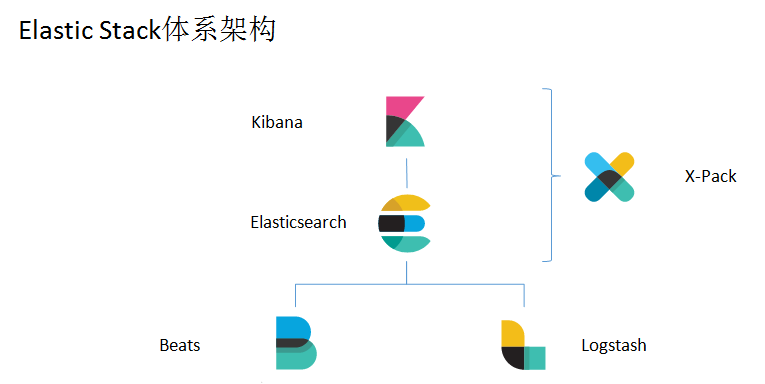
     当务之急我们使用集中化的日志管理，例如：开源的 syslog ，将所有服务器上的日志收集汇总。

     集中化管理日志后，日志的统计和检索又成为一件比较麻烦的事情，一般我们使用 grep 、 awk和 wc 等 Linux 命令能实现检索和统计，

     但是对于要求更高的查询、排序和统计等要求和庞大的机器数量依然使用这样的方法难免有点力不从心。

     开源实时日志分析 ELK 平台能够完美的解决我们上述的问题， ELK 由 ElasticSearch 、 Logstash 和 Kiabana 三个开源工具组成。

**3、体系架构**





工作原理如下如所示：

在需要收集日志的所有服务上部署 logstash ，其中 logstash agent （ logstash shipper ）用于监控并过滤收集日志，将过滤后的内容发送到 logstash indexer ， logstash indexer 将日志收集在一起交给全文搜索服务 ElasticSearch ，

可以用 ElasticSearch 进行自定义搜索

通过 Kibana 来结合自定义搜索进行页面展示。

**3、elk的技术架构**

     gateway：hdfs、Amazon S3、Local FileSystem、Shared FileSystem

     Index module、Search module、 Mapping、River（引入异构数据的插件机制：RabbitMQ-River、Twitter-River）

     zen和ec2---zk、scripting（mvel、python、js等）

     第三方插件（Head、）

     Transport（Thrift、Memcached、Http）

     Java（Netty）

     restful 和 curl

**4、elk的基本概念**

     node 和 cluster

**index（数据库）->type（表）->document（行）->field （字段）**

     shards 分片

     replicas 复制

**5、其他全文搜索**

     Solr：文本支持强，html、pdf、word、excel、cvs

     Elasticsearch：实时数据分析，支持json格式

     splunk：

**6、resftful风格**

**7、curl**

     curl www.baidu.com

     curl -o baidu.txt www.baidu.com

     curl -O www.baidu.com/xxx.html

     curl -i   www.baidu.com 显示HTTP头信息

     curl -v  www.baidu.com 显示过程

     curl -X GET/PUT/POST/DELETE www.baidu.com

**8、查看官方api**

     单模式：

     多模式：

# Apache kafka：一个高吞吐量的分布式发布订阅消息系统

        从最初的了解、学习kafka，到后来基于其API进行开发，再到后来有意识地了解其实现原理，是一个漫长的过程IMG_256。趁周末有空梳理一下期间读过的官方文档、技术博客等，以便自己日后复习。

特性如下：

* Fast：单台broker节点每秒便可处理来自数千个client的(hundreds of megabytes of reads and writes)百兆读写请求；
* Scalable：扩展性：可以实现灵活、透明的扩展without downtime 。日志数据被分区并分布式地存储在集群的服务器中，允许数据量大于任意单个节点的存储能力，并且消费者也是一个可扩展的集群。
* Durable：持久化：消息被持久化在磁盘Disk中，且在集群内进行备份避免单点故障导致消息丢失。每个broker节点可存储TB级消息；
* Distributed by Design：分布式设计：kafka采用cluster-centric设计，具备很强的持久性和容错保证；

Getting Started

kafka是一个分布式的、分区的、存在备份的commit log service。

消息分类存储，每一类称之为Topic；发布消息的叫做生产者、订阅消费消息的是消费者；

1. kafka体现为多台服务器组成集群，每一个server也叫broker；
2. client与server之间使用TCP协议进行通信（采用TCP socket）；
3. topic又被划分为若干分区partition（创建topic的时候可以指定分区个数），每个分区由一组有序的消息序列组成，新生产的消息则追加至末尾。
4. 每条消息有一个序列号id，称之为offset，唯一标记partition内的一条消息；
5. kafka集群会保存所有消息，无论其是否被消费过，可以在配置文件中设置保存的时间以及日志清理策略（delete或者compact）；
6. 消费者需要维护的唯一元数据信息是其消费的offset值，理论上offset是线性增长的，但consumer可以指定offset的值，所以这些设计机制使consumer very cheap，开发者可以随意增删consumer，不会影响kafka集群或者其他消费者，broker也是无状态的；
7. 分区partition存在的意义：将一个topic的消息进行切分，分开存储在不同的broker上，从而使topic能够容纳更多的数据量，不受单台服务器存储能力的限制；另外一点是：作为并行处理的基本单元：（我的理解：1、一个partition对应一个leader broker以及若干follower，生产者消费者仅和leader通信进行读写操作，只要集群内的leader均匀分布，则可保证每台服务器的负载相对均衡；2、消费者又属于某个consumer group，一条消息只能被一个group内的一个consumer消费，所以一般情况下，一个消费者消费topic的一个分区，可见分区越多，便可以有更多的消费者并发消费，所以partition是kafka并行化的基本单位）
8. 消费者：
   * 传统的消息系统有两种模型：队列queuing和发布订阅publish-subscribe。队列模型中，一条消息只能被一个消费者消费；发布订阅模型中，一条消息会广播给所有的消费者。kafka通过consumer group能同时满足这两种需求。
   * 消息的顺序性保证：传统的队列保证消息在server上顺序存储，但当他将消息按照存储顺序分发给并行消费的consumer时，顺序就会被打乱，无法保障有序性。当然，可以只让一个消费者来消费，这样消息依然是顺序的，但又违背了并行处理的初衷。
   * kafka通过partition很好的保证了顺序性以及消费者的负载均衡：（这是partition存在的另一个意义吧）consumer group中的一个consumer消费一个partition，所以保证了一个分区内的消息总是被顺序消费，消费者的负载也相对均衡。但是要注意：如果group内的consumer数量大于topic的partition个数，则一些消费者处于空闲状态，而且不能保证partition之间消息的顺序性，特殊场景下，可以使一个topic只有一个分区，只让一个consumer进行消费，这样便是全局有序的。

# 大型网站架构技术一览（系统性能、可用性、伸缩性、扩展性、安全性）

2014-09-29  [且看且珍惜](http://www.360doc.com/userhome/16650130)   阅 2352  转 40

[转藏到我的图书馆微信](http://www.360doc.com/content/14/0929/21/javascript:void(0);)分享：

|  |
| --- |
| 网站系统架构层次：前端架构、应用层架构、服务层架构、存储层架构、后台架构、数据采集与监控、安全架构、数据中心机房架构。  1.前端架构（浏览器优化技术、CDN、动静分离，静态资源独立部署、图片服务、反向代理、DNS） 前端指用户请求到达网站应用服务器之前经历的环节，通常不包含网站业务逻辑，不处理动态内容。  浏览器优化技术 并不是优化浏览器，而是通过优化响应页面，加快浏览器页面的加载和显示，常用的有页面缓存、合并HTTP减少请求次数、使用页面压缩等。  CDN 内容分发网络，部署在网络运营商机房，通过将静态页面内容分发到离用户最近最近的CDN服务器，使用户可以通过最短路径获取内容。  动静分离，静态资源独立部署 静态资源，如JS、CSS等文件部署在专门的服务器集群上，和Web应用动态内容服务分离，并使用专门的（二级）域名。  图片服务 图片不是指网站Logo、按钮图标等，这些文件属于上面提到的静态资源，应该和JS、CSS部署在一起。这里的图片指用户上传的图片，如产品图片、用户头像等，图片服务同样适用独立部署的图片服务器集群，并使用独立（二级）域名。  反向代理 部署在网站机房，在应用服务器、静态资源服务器、图片服务器之前，提供页面缓存服务。  DNS 域名服务，将域名解析成IP地址，利用DNS可以实现DNS负载均衡，配置CDN也需要修改DNS，使域名解析后指向CDN服务器。  2.应用层架构（开发框架、页面渲染、负载均衡、Session管理、动态页面静态化、业务拆分、虚拟化服务器） 应用层是处理网站主要业务逻辑的地方。  开发框架 网站业务是多变的，网站的大部分软件工程师都是在加班加点开发网站业务，一个好的开发框架至关重要。一个号的开发框架应该能够分离关注面，使美工、开发工程师可以各司其事，易于协作。同时还应该内置一些安全策略，防护Web用攻击。  页面渲染 将分别开发维护的动态内容和静态页面模板集成起来，组合成最终显示给用户的完整页面。  负载均衡 将多台应用服务器组成一个集群，通过负载均衡技术将用户请求分发到不同的服务器上，以应对大量用户同时访问时产生的高并发负载压力。  Session管理 为了实现高可用的应用服务器集群，应用服务器通常设计为无状态，不保存用户请求上下文信息，但是网站业务通常需要保持用户会话信息，需要专门的机制管理Session，使集群内甚至跨集群的应用服务器可以共享Session。  动态页面静态化 对于访问量特别大而更新又不很频繁的动态页面，可以将其静态化，即生成一个静态页面，利用静态页面的优化手段加速用户访问，如反向代理、CDN、浏览器缓存等。  业务拆分 将复杂而庞大的业务拆分开来，形成多个规模较小的产品，独立开发、部署、维护，除了降低系统耦合度，也便于数据库业务分库。按业务对关系数据库进行拆分，技术难度相对较小，而效果又相对较好。  虚拟化服务器 将一台物理服务器虚拟化成多态虚拟服务器，对于并发访问较低的业务，更容易用较少的资源构架高可用的应用服务器集群。  3.服务层架构（分布式消息、分布式服务、分布式缓存、分布式配置） 提供基础服务，供应用层调用，完成网站业务。  分布式消息 利用消息队列机制，实现业务和业务、业务和服务之间的异步消息发送及低耦合的业务关系。  分布式服务 提供高性能、低耦合、易复用、易管理的分布式服务，在网站实现面向服务架构（SOA）。  分布式缓存 通过可伸缩的服务器集群提供大规模热点数据的缓存服务，是网站性能优化的重要手段。  分布式配置 系统运行需要配置许多参数，如果这些参数需要修改，比如分布式缓存集群加入新的缓存服务器，需要修改应用程序客户端的缓存服务器列表配置，并重启应用程序服务器。分布式配置在系统运行期提供配置动态推送服务，将配置修改实时推送到应用系统，无需重启服务器。  4.存储层架构（分布式文件、关系数据库、NoSQL数据库、数据同步） 提供数据、文件的持久化存储访问与管理服务。  分布式文件 网站在线业务需要存储的文件大部分都是图片、网页、视频等比较小的文件，但是这些文件的数量非常庞大，而且通常都在持续增加，需要伸缩性设计比较好的分布式文件系统。   关系数据库 大部分万丈的主要业务是基于关系数据库开发的，但是关系数据库对集群伸缩性的支持表较差。通过在应用程序的数据访问层增加数据库访问的路由功能，根据业务配置将数据库访问路由到不同的物理数据库上，可实现关系数据库的分布式访问。  NoSQL数据库 目前各种NoSQL数据库层出不穷，在内存管理、数据模型、集群分布式管理等方面各有优势，不过从社区活动性角度看，HBase无疑是目前最好的。  数据同步 在支持全球范围内数据共享的分布式数据库技术成熟之前，拥有多个数据中心的网站必须在多个数据中心之间进行数据同步，以保证每个数据中心都拥有完整的数据。在实践中，为了减轻数据库压力，将数据库的事物日志（或者NoSQL的写操作Log）同步到其他数据中心，根据Log进行数据重演，实现数据同步。  5.后台架构（搜索引擎、数据仓库、推荐系统） 网站应用中，除了要处理用户的实时访问请求外，还有一些后台非实时数据分析要处理。  搜索引擎 即使是网站内部的搜索引擎，也需要进行数据增量更新及全量更新、构建索引等。这些操作通过后台系统定时执行。  数据仓库 根据离线数据，提供数据分析与数据挖掘服务。  推荐系统 社交网站及购物网站通过挖掘人与人之间的关系，人和商品之间的关系，发展潜在的人际关系和购物兴趣，为用户提供个性化推荐服务。  6.数据采集于监控（浏览器数据采集、服务器业务数据采集、服务器性能数据采集、系统监控、系统报警） 监控网站访问情况与系统运行情况，为网站运营决策和运维管理提供支持保障。  浏览器数据采集 通过在网站页面中嵌入JS脚本采集用户浏览器环境与操作记录，分析用户行为。  服务器业务数据采集 服务器业务数据包括两种，一种是采集在服务器端记录的用户请求操作日志；一种是采集应用程序运行期业务数据，比如待处理消息数目等。  服务器性能数据采集 采集服务器性能数据，如系统负载、内存使用率、网卡流量等。  系统监控 将前述采集的数据以图表的方式展示，以便运营和运维人员监控网站运行状况，做到这一步仅仅是系统监视。更先进的做法是根据采集的数据进行自动化运维，自动处理系统异常状况，是吸纳自动化控制。  系统报警 如果采集来的数据超过预设的正常情况的阀值，比如系统负载过高，就通过邮件、短信、语音电话等方式发出警报信号，等待工程师干预。  7.安全架构（Web攻击、数据保护） 保护网站免遭攻击及敏感信息泄露。  Web攻击 以HTTP请求的方式发起的攻击，危害最大的就是XSS和SQL注入攻击。但是只要措施得当，这两种攻击都是比较容易防范的。  数据保护 敏感信息加密传输与存储，保护网站和用户资产。  8.数据中心机房架构（机房架构、机柜架构、服务器架构） 大型网站需要的服务器规模数以十万计，机房物理架构也需要关注。  机房架构 对于一个拥有十万台服务器的大型网站，每台服务器耗电（包括服务器本身耗电及空调耗电）每年大约需要人民币2000元，那么网站每年机房电费就需要两亿人民币。数据中心能耗问题日趋严重，Google、Facebook选择数据中心地理位置的时候趋向选择散热良好，供电充裕的地方。  机柜架构 包括机柜大小，网线布局、指示灯规格、不间断电源、电压规格（是48V直流电还是220V民用交流电）等一系列问题。  服务器架构 大型网站由于服务器采购规模庞大，大都采用定制服务器的方式代替购买服务器整机。根据网站应用需求，定制硬盘、内存、甚至CPU，同时去除不必要的外设接口（显示器输出接口，鼠标、键盘输入接口），并使空间结构利于散热。 |

**构建微服务：Spring boot 入门篇**

# ****什么是****spring boot

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。用我的话来理解，就是spring boot其实不是什么新的框架，它默认配置了很多框架的使用方式，就像maven整合了所有的jar包，spring boot整合了所有的框架（不知道这样比喻是否合适）。

# ****使用****spring boot****有什么好处****

其实就是简单、快速、方便！平时如果我们需要搭建一个spring web项目的时候需要怎么做呢？

1）配置web.xml，加载spring和spring mvc

2）配置数据库连接、配置spring事务

3）配置加载配置文件的读取，开启注解

4）配置日志文件

...

配置完成之后部署tomcat 调试

...

现在非常流行微服务，如果我这个项目仅仅只是需要发送一个邮件，如果我的项目仅仅是生产一个积分；我都需要这样折腾一遍!

但是如果使用spring boot呢？

很简单，我仅仅只需要非常少的几个配置就可以迅速方便的搭建起来一套web项目或者是构建一个微服务！

使用sping boot到底有多爽，用下面这幅图来表达

[](http://images2015.cnblogs.com/blog/331425/201607/331425-20160712105132873-2014012760.jpg)

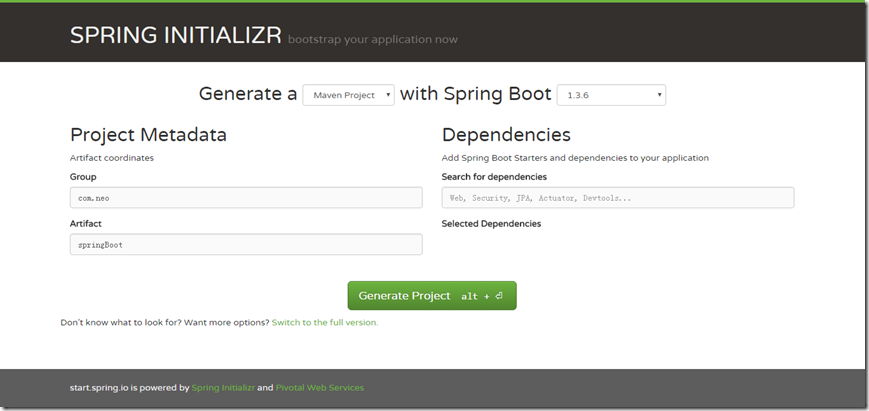
# ****快速入门****

说了那么多，手痒痒的很，马上来一发试试!

**maven构建项目**

1、访问http://start.spring.io/

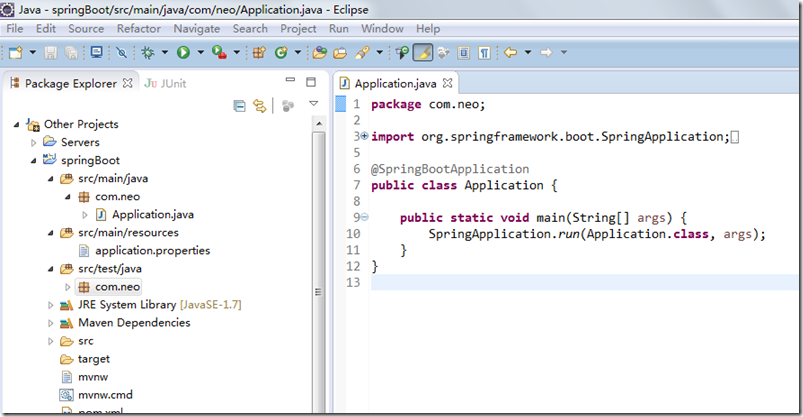
2、选择构建工具Maven Project、Spring Boot版本1.3.6以及一些工程基本信息，点击“[Switch to the full version.](http://start.spring.io/)”java版本选择1.7，可参考下图所示：

[](http://images2015.cnblogs.com/blog/331425/201607/331425-20160712105134607-1615281203.png)

3、点击Generate Project下载项目压缩包

4、解压后，使用eclipse，Import -> Existing Maven Projects -> Next ->选择解压后的文件夹-> Finsh，OK done!

**项目结构介绍**

[](http://images2015.cnblogs.com/blog/331425/201607/331425-20160712105137014-669859839.png)

如上图所示，Spring Boot的基础结构共三个文件:

src/main/java  程序开发以及主程序入口

src/main/resources 配置文件

src/test/java  测试程序

另外，spingboot建议的目录结果如下：

root package结构：com.example.myproject

[IMG_259](http://www.cnblogs.com/ityouknow/p/javascript:void(0);)

# angularjs

以前开发（web或者移动端）前端主要使用jQuery+原生js,如果使用某些前端UI框架的话，它自己还可能提供一些API可以使用。而且目前很多UI框架都是基于jQuery的，所以说一下由jQuery跨到angularjs跨度较大，研究了一段时间的angularjs ,下面从整体上说说感受吧:

**关于和jquery的比较**

首先angular是一个mvc框架，它与jquery不同之处在于，前者致力于mvc代码解耦，采用model,controller以及view方式去组织代码，而后者提供给你了很多APi函数，你可以不用写很多原生js去实现比较复杂的效果，比如说动画，$.animate,这样的效果如果需要原生js来写的话，代码量将会比较庞大；

其次，jQuery没有定义你的代码如何组织，你可以将它放在一个单独的js文件中进行引用，也可以直接写在页面中采用script标签进行包裹，甚至可以直接以内联的方式写在html标签中，但是angularjs会将一个HTML页面分成若干个模块，每个模块都可以自己的scope，service以及directive，各个模块之间也可以进行通信，但是整体上结构是比较清晰的，就是说其代码组织方式是模块化的。

最后，jQuery的思想是先设计好页面，然后在已有页面的基础上进行dom操作后展示页面，但是angular的view可能仅仅是一个框架，对view的dom操作或者时间监听都是在directive中实现的，而且一般情况下很少自己直接去写Dom操作代码，只要你监听model。model发生变化后view也会发生变化。

**关于适用场合**

jQuery应该适用于大多数web开发，移动端也有（jQuerymobile），angularjs有人说更适合做SPA（我个人认为在手机上的SPA可能会引发性能上的问题，因为它的脏检查机制会影响性能），在web端，一些CRUD的应用或者管理类软件还是可以使用的（当然这里的理解可能不一定准确，会随着深入学习更多去了解和使用）。

**关于UI的结合**

开发任何产品都需要用到前端UI，目前很多UI是基于jQuery的，这意味着你如果要用angularjs和这些Ui组件的话，需要用angularjs的directive去重写些组件，这一过程是比较麻烦的，所幸的是，angular给我们提供了一些UI组件可以使用（web端主要是结合bootstrap前端组件），<http://angular-ui.github.io/>，而在移动端主要是结合ionic框架<http://ionicframework.com/>，但是随着angular的发展，很多HTML5的前端框架也慢慢集成了angularjs版本可供使用。

**关于angularjs的特点**

1.数据的双向绑定：这可能是其最激动人心的特性吧，view层的数据和model层的数据是双向绑定的，其中之一发生更改，另一方会随之变化，这不用你写任何代码！（想想jQuery方式下怎么做吧）

2.代码模块化，每个模块的代码独立拥有自己的作用域，model，controller等。

3.强大的directive可以将很多功能封装成HTML的tag,属性或者注释等，这大大美化了HTML的结构，增强了可阅读性；

4.依赖注入，将这种后端语言的设计模式赋予前端代码，这意味着前端的代码可以提高重用性和灵活性，未来的模式可能将大量操作放在客户端，服务端只提供数据来源和其他客户端无法完成的操作；

5.测试驱动开发，angularjs一开始就以此为目标，使用angular开发的应用可以很容易地进行单元测试和端对端测试，这解决了传统的js代码难以测试和维护的缺陷

以上就是研究angularjs一段时间得出的结论，其中某些地方可能有所疏漏，没关系，接下来会展开其中某一点一步步去学习。

# logstash采集与清洗数据到elasticsearch案例实战

原创 2016年10月28日 10:13:03

标签：

[elasticsearch](http://so.csdn.net/so/search/s.do?q=elasticsearch&t=blog) /

[logstash](http://so.csdn.net/so/search/s.do?q=logstash&t=blog) /

[数据清洗](http://so.csdn.net/so/search/s.do?q=%E6%95%B0%E6%8D%AE%E6%B8%85%E6%B4%97&t=blog)

1334

Logstash的使用

logstash支持把配置写入文件 xxx.conf，然后通过读取配置文件来采集数据  
./bin/logstash –f xxx.conf  
  
logstash最终会把数据封装成json类型，默认会添加@timestamp时间字段、host主机字段、type字段。原消息数据会整个封装进message字段。如果数据处理过程中，用户解析添加了多个字段，则最终结果又会多出多个字段。也可以在数据处理过程中移除多个字段，总之，logstash最终输出的数据格式是json格式。

Logstash的结构

Logstash由 input，filter，output三个组件去完成采集数据  
如下是一个logstash的配置实例：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

input {

    file {

        type => "log"

        path => "/log/\*/\*.log"

        discover\_interval => 10

        start\_position => "beginning"

    }

}

filter {

}

output {

    elasticsearch {

    index => "log-%{+YYYY.MM.dd}"

    hosts => ["172.16.0.14:9200", "172.16.0.15:9200", "172.16.0.16:9200"]

    }

    stdout {codec => rubydebug}

}

input

input组件负责读取数据，可以采用file插件读取本地文本文件，stdin插件读取标准输入数据，tcp插件读取网络数据，log4j插件读取log4j发送过来的数据等等。

filter

filter插件负责过滤解析input读取的数据，可以用grok插件正则解析数据，date插件解析日期，json插件解析json等等。

output

output插件负责将filter处理过的数据输出。可以用elasticsearch插件输出到es，rediss插件输出到redis，stdout插件标准输出，kafka插件输出到kafka等等  
trade.log日志采集。

trade.log日志采集

配置内容如下：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

input {

    file {

        type => "tradelog"

        path => "/home/elk/his/trade.log\*"

        discover\_interval => 5

        start\_position => "beginning"

        sincedb\_path => "/home/elk/myconf/sincedb\_trade.txt"

        sincedb\_write\_interval => 15

        codec => plain { charset => "GB2312" }

    }

}

filter {

    grok {

        match => { "message" => "%{DATESTAMP\_CN:[@metadata][logdate]} .\* - %{WORD:opeType}\|%{WORD:name}\|Oid: %{WORD:oid}\|IP: %{IP:ip}\|MAC: %{GREEDYDATA:mac}\|%{WORD:result}\|%{GREEDYDATA:exception}\|" }

        match => { "message" => "%{DATESTAMP\_CN:[@metadata][logdate]} .\* - %{WORD:opeType}\|%{WORD:name}\|Oid: %{WORD:oid}\|IP: %{IP:ip}\|MAC: %{GREEDYDATA:mac}\|%{WORD:result}\|"  }

        match => { "message" => "%{DATESTAMP\_CN:[@metadata][logdate]} .\* - %{WORD:opeType}\|%{WORD:name}\|Oid: %{WORD:oid}\|IP: %{IP:ip}\|MAC: %{GREEDYDATA:mac}\|"  }

        match => { "message" => "%{DATESTAMP\_CN:[@metadata][logdate]} .\* - %{WORD:opeType}\|IP: %{IP:ip}\|MAC: %{GREEDYDATA:mac}\|%{WORD:result}\|"  }

        match => { "message" => "%{DATESTAMP\_CN:[@metadata][logdate]} .\* - %{WORD:opeType}\|IP: %{IP:ip}\|MAC: %{GREEDYDATA:mac}\|" }

        remove\_field  => "message"

    }

    date {

        match => ["[@metadata][logdate]", "YYYY-MM-dd HH:mm:ss,SSS"]

    }

}

output {

    if "\_grokparsefailure" not in [tags] and "\_dateparsefailure" not in [tags] {

        stdout {codec => rubydebug}

        elasticsearch {

            index => "log4j-tradelog"

            hosts => ["168.7.1.67:9200"]

            manage\_template => true

            template\_overwrite => true

            template\_name => "log4j-tradelog"

            template => "/home/elk/myconf/tradelog\_template.json"

        }

    }

}

input

1.    start\_position：设置beginning保证从文件开头读取数据。  
2.    path：填入文件路径。  
3.    type：自定义类型为tradelog，由用户任意填写。  
4.    codec：设置读取文件的编码为GB2312,用户也可以设置为UTF-8等等  
5.    discover\_interval：每隔多久去检查一次被监听的 path 下是否有新文件，默认值是15秒  
6.    sincedb\_path：设置记录源文件读取位置的文件，默认为文件所在位置的隐藏文件。  
7.    sincedb\_write\_interval：每隔15秒记录一下文件读取位置

filter

日志格式如下：

[plain] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

2016-05-09 09:49:13,817 [] [ACTIVE] ExecuteThread: '1' for queue: 'weblogic.kernel.Default (self-tuning)' [INFO ] com.c.command.StartLogCommand.execute(StartLogCommand.java:46) - FrontPageFinProdListQry|IP: 192.168.1.105|MAC: A1345C05-26C1-4263-8845-01CFCA6EC4FD|

2016-05-09 09:49:13,928 [] [ACTIVE] ExecuteThread: '2' for queue: 'weblogic.kernel.Default (self-tuning)' [INFO ] com.c.command.EndLogCommand.execute(EndLogCommand.java:44) - FrontPageAdvertListQry|IP: 192.168.1.105|MAC: A1245C05-26C1-4263-8845-01CFCA6EC4FD|Success|

grok插件

因为该日志中有5种格式如下，以最后几个我需要的字段为例说明：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

交易名|登录名|编号|ip地址|mac地址|返回结果|异常信息

交易名|登录名|编号|ip地址|mac地址|返回结果|

交易名|登录名|编号|ip地址|mac地址|

交易名|ip地址|mac地址|返回结果|

交易名|ip地址|mac地址|

所以采用5种正则规则去匹配，logstash默认会从上到下按规则去匹配，直到匹配上为止。（日志中的多行错误信息，匹配不上，logstash会在tags字段添加”\_ grokparsefailure”，所以后面输出的时候会用if条件判断过滤掉解析失败的行消息）

注意：5种正则规则的上下顺序，下面的规则放在上面会导致可能内容解析不全，比如源数据是：请求交易名|操作员登录名|操作员编号|ip地址|mac地址|返回结果|异常信息，如果按照“请求交易名|ip地址|mac地址|”规则去匹配，只能识别出3个字段，而且匹配成功，不继续往下执行，这样识别的内容就不全。  
  
logstash内置了很多正则匹配规则，用户可以直接调用这些规则来解析，例如%{WORD:result} 表示调用WORD规则（即识别字符串规则）来解析并最后赋值给result字段（result字段会自动创建）。  
    下面以第一条match规则为例来说明：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

match => { "message" => "%{DATESTAMP\_CN:[@metadata][logdate]} .\* - %{WORD:opeType}\|%{WORD:name}\|Oid: %{WORD:oid}\|IP: %{IP:ip}\|MAC: %{GREEDYDATA:mac}\|%{WORD:result}\|%{GREEDYDATA:exception}\|" }

首先行首使用DATESTAMP\_CN规则来识别时间，并赋值给logdate字段名；然后.\*识别任意字符串（.代表任意一个字符，包括特殊字符，\*代表个数是任意个）；然后使用WORD规则（即匹配字符串规则，不包含特殊字符）识别到字符串并赋值给opeType字段；后面同理。这些WORD、IP、GREEDYDATA规则都是logstash内部grok-patterns文件已经定义好了的规则。用户可以直接拿来使用。  
  
注意：[@metadata]表示logdate这个字段在数据处理过程中只是一个临时字段，最后不会真的输出。避免了使用remove\_field手动移除字段。  
  
注意：logstash默认不支持”YYYY-MM-dd HH:mm:ss,SSS”格式的时间匹配，需要自己定义正则表达式到logstash-2.3.1/vendor/bundle/jruby/1.9/gems/logstash-patterns-core-2.0.5/patterns/grok-patterns文件中。grok-patterns文件中追加2行内容：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

DATE\_CN %{YEAR}[./-]%{MONTHNUM}[./-]%{MONTHDAY}

DATESTAMP\_CN %{DATE\_CN} %{TIME}

注意：logstash的正则表达式采用ruby语言正则表达式，具体语法可以参考网上。  
  
remove\_field  => "message"表示解析完成之后删除原来的 message字段，避免重复。

date插件

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

match => ["[@metadata][logdate]", "YYYY-MM-dd HH:mm:ss,SSS"]

logstash默认的时间字段是@timestamp,如果不设置的话，默认是数据采集时候的时间，这里我们将日志打印的时间（即解析出的logdate字段的内容）设置为@timestamp内容，方便之后kibana根据时间检索。  
  
注意：解析出来的@timestamp会比实际时间早8个小时，这是内置utc时间格式问题，kibana页面展示的时候会根据浏览器当前时区自动转换回来，这里不作处理。

output

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

if "\_grokparsefailure" not in [tags] and "\_dateparsefailure" not in [tags] {

    stdout {codec => rubydebug}

    elasticsearch {

        index => "log4j-tradelog"

        hosts => ["134.7.1.67:9200"]

        manage\_template => true

        template\_overwrite => true

        template => "/home/elk/myconf/tradelog\_template.json"

    }

}

前面提到过，如果grok解析失败，会在tags字段自动添加\_grokparsefailure值，如果date解析失败，会在tags字段自动添加\_dateparsefailure值。所以最后的输出，我们采用条件过滤掉解析失败的行内容。最终的每一行内容解析成json，一路存入elasticsearch，另一路进行标准输出。

elasticsearch插件

index：要导入的es索引  
host：es地址，有多个节点配置多个节点  
template：指定elasticsearch的mapping模板文件，如果该索引不存在，logstash会根据这个mapping模板去自动创建索引。

stdout插件

rubydebug标准输出，便于调试，可以不使用该插件。  
  
最终解析出结果示例如下：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

{

      "@version" => "1",

    "@timestamp" => "2016-05-09T01:44:48.366Z",

          "path" => "/home/elk/e.log",

          "host" => "ccc7",

          "type" => "tradelog",

       "opeType" => "WZQry",

          "name" => "lhcsssz2",

           "oid" => "abzzak",

            "ip" => "192.168.44.105",

           "mac" => "A1345C05-26C1-4253-8845-01CFCA8EC4FD",

        "result" => "Success"

}

error.log采集

日志实例：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

2016-09-29 17:13:24,184|ncid=1100343164|oid=acaatv|loginName=zhenglw1|transId=Withdraw|traceId=N/A-\_A-88C4D-043|exceptType=com.intenft.exception.AppRTException|exceptCode=CORESYST\_TXN\_NATIVE\_89042|exceptMsg=对不起！记录没有找到

配置文件如下：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

input {

    file {

        path => "/home/elk/his/error.log\*"

        type => "errorlog"

        start\_position => "beginning"

        discover\_interval => 5

        codec => multiline {

            charset => "GB2312"

            pattern => "^%{DATESTAMP\_CN}"

            negate => true

            what => "next"

        }

        sincedb\_path => "/home/elk/myconf/sincedb\_error.txt"

        sincedb\_write\_interval => 15

    }

}

filter {

    grok {

        match => { "message" => "%{DATESTAMP\_CN:[@metadata][logdate]}%{GREEDYDATA:[@metadata][keyvalue]}" }

        remove\_field  => "message"

    }

    date {

        match => ["[@metadata][logdate]", "YYYY-MM-dd HH:mm:ss,SSS"]

    }

    kv {

        source => "[@metadata][keyvalue]"

        field\_split => "\|"

        value\_split => "="

    }

}

output {

    if "multiline" in [tags] {

        stdout {codec => rubydebug}

        elasticsearch {

            index => "log4j-errorlog-3"

            hosts => ["168.7.1.67:9200"]

            manage\_template => true

            template\_overwrite => true

            template => "/home/elk/myconf/errorlog\_template.json"

        }

    }

}

input

8.    start\_position：设置beginning保证从文件开头读取数据。  
9.    path：填入文件路径。  
10.    type：自定义类型为tradelog，由用户任意填写。  
11.    codec：multiline插件   
12.    discover\_interval：每隔多久去检查一次被监听的 path 下是否有新文件，默认值是15秒  
13.    sincedb\_path：设置记录源文件读取位置的文件，默认为文件所在位置的隐藏文件。  
14.    sincedb\_write\_interval：每隔15秒记录一下文件读取位置

multiline插件

logstash默认读取一行内容为一个消息，因为错误日志包含堆栈信息，多行对应一个消息，所以使用该插件合并多行为一条消息。  
pattern：以”YYYY-MM-dd HH:mm:ss,SSS”格式开头的匹配为一条消息。  
negate：true 表示正向使用该patttern  
what：匹配到的日期属于下一条消息  
charset：设置文件编码

filter

grok插件

匹配日期到logdata字段，匹配剩下的所有字符串到keyvalue临时字段，”GREEDYDATA”正则表达式为”.\*”

date插件

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

match => ["[@metadata][logdate]", "YYYY-MM-dd HH:mm:ss,SSS"]

logstash默认的时间字段是@timestamp,如果不设置的话，默认是数据采集时候的时间，这里我们将日志打印的时间（即解析出的logdate字段的内容）设置为@timestamp内容，方便之后kibana根据时间检索。  
  
注意：解析出来的@timestamp会比实际时间早8个小时，这是内置utc时间格式问题，kibana页面展示的时候会根据浏览器当前时区自动转换回来，这里不作处理。

kv插件

source：解析前面grok获取的keyvalue字段

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

（比如：|ncid=1100783164|oid=acaatv|loginName=zhew1|transId=Withdraw|traceId=N/A-\_A-88C4D-043|exceptType=com.inteft.exception.AppRTException|exceptCode=CORESYST\_TXN\_NATIVE\_89042|exceptMsg=对不起！记录没有找到）

field\_split：按”|”切分key-value对  
value\_split：按”=”切分key 和 value，最终切分出来key作为字段名，value作为字段值

output

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

output {

    if "multiline" in [tags] {

        stdout {codec => rubydebug}

        elasticsearch {

            index => "log4j-errorlog-3"

            hosts => ["168.7.1.67:9200"]

            manage\_template => true

            template\_overwrite => true

            template => "/home/elk/myconf/errorlog\_template.json"

        }

    }

}

该日志有2种格式的日志，一种是单行的错误信息日志，一种是多行的包含堆栈信息的日志，这2种日志内容重复，那么只需要解析单行格式的日志。kv插件解析多行格式的日志时， tags字段里没有”multipline”值（原因是因为grok解析的时候不能解析换行符），所以可以通过if条件判断tags字段是否有”multipline”值，来过滤掉多行格式的日志。

elasticsearch插件

index：要导入的es索引  
host：es地址，有多个节点配置多个节点  
template：指定elasticsearch的mapping模板文件，如果该索引不存在，logstash会根据这个mapping模板去自动创建索引。  
  
最终解析的结果示例如下：

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

{

    "@timestamp" => "2016-09-29T09:14:22.194Z",

      "@version" => "1",

          "tags" => [

        [0] "multiline"

    ],

          "path" => "/home/elk/stst.log",

          "host" => "ci7",

          "type" => "sttlog",

          "ncid" => "1143164",

           "oid" => "acav",

     "loginName" => "zhew1",

       "transId" => "MyQuery",

       "traceId" => "N/A8C4E-047",

    "exceptType" => "com.exception.AppRTException",

    "exceptCode" => "CORESYNATIVE\_82243",

     "exceptMsg" => "对不起！根据账号获取客户信息错误"

}

总结：

注意：

logstash filter中的每一个插件都有add\_field,remove\_field,add\_tag,remove\_tag  4个功能。

附录：

mapping模板文件

tradelog:

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

{

    "template": "log4j-tradelog\*",

    "settings": {

        "index.number\_of\_shards": 3,

        "number\_of\_replicas": 0

    },

    "mappings": {

        "tradelog": {

            "\_all": {

                "enabled": false

            },

            "properties": {

                "@timestamp": {

                    "type": "date",

                    "format": "strict\_date\_optional\_time||epoch\_millis",

                    "doc\_values": true

                },

                "@version": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "exception": {

                    "type": "string",

                    "index": "analyzed"

                },

                "path": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "host": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "ip": {

                    "type": "ip",

                    "index": "not\_analyzed"

                },

                "logger\_name": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "mac": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "name": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "oid": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "opeType": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "priority": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "result": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "type": {

                    "type": "string",

                    "index": "not\_analyzed"

                }

            }

        }

    }

}

errorlog:

[ruby] [view plain](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1) [copy](http://blog.csdn.net/qq1032355091/article/details/52953837?locationNum=3&fps=1)

{

    "template": "log4j-errorlog\*",

    "settings": {

        "index.number\_of\_shards": 3,

        "number\_of\_replicas": 0

    },

    "mappings": {

        "errorlog": {

            "\_all": {

                "enabled": false

            },

            "properties": {

                "host": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "ncid": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "type": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "@version": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "exceptType": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "@timestamp": {

                    "format": "strict\_date\_optional\_time||epoch\_millis",

                    "type": "date"

                },

                "exceptCode": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "transId": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "priority": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "oid": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "traceId": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "exceptMsg": {

                    "type": "string",

                    "index": "analyzed"

                },

                "path": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "logger\_name": {

                    "type": "string",

                    "index": "not\_analyzed"

                },

                "loginName": {

                    "type": "string",

                    "index": "not\_analyzed"

                }

            }

        }

    }

}

# [Netty](http://lippeng.iteye.com/blog/1907279)

**博客分类：**

* [Thinking](http://lippeng.iteye.com/category/102425)

Netty是什么？

本质：JBoss做的一个Jar包

目的：快速开发高性能、高可靠性的网络服务器和客户端程序

优点：提供异步的、事件驱动的网络应用程序框架和工具

通俗的说：一个好使的处理Socket的东东

如果没有Netty？

远古：java.net + java.io

近代：java.nio

其他：Mina，Grizzly

为什么不是Mina？

1、都是Trustin Lee的作品，Netty更晚；

2、Mina将内核和一些特性的联系过于紧密，使得用户在不需要这些特性的时候无法脱离，相比下性能会有所下降，Netty解决了这个设计问题；

3、Netty的文档更清晰，很多Mina的特性在Netty里都有；

4、Netty更新周期更短，新版本的发布比较快；

5、它们的架构差别不大，Mina靠apache生存，而Netty靠jboss，和jboss的结合度非常高，Netty有对google protocal buf的支持，有更完整的ioc容器支持(spring,guice,jbossmc和osgi)；

6、Netty比Mina使用起来更简单，Netty里你可以自定义的处理upstream events 或/和 downstream events，可以使用decoder和encoder来解码和编码发送内容；

7、Netty和Mina在处理UDP时有一些不同，Netty将UDP无连接的特性暴露出来；而Mina对UDP进行了高级层次的抽象，可以把UDP当成"面向连接"的协议，而要Netty做到这一点比较困难。

Netty的特性

设计

统一的API，适用于不同的协议（阻塞和非阻塞）

基于灵活、可扩展的事件驱动模型

高度可定制的线程模型

可靠的无连接数据Socket支持（UDP）

性能

更好的吞吐量，低延迟

更省资源

尽量减少不必要的内存拷贝

安全

完整的SSL/TLS和STARTTLS的支持

能在Applet与Android的限制环境运行良好

健壮性

不再因过快、过慢或超负载连接导致OutOfMemoryError

不再有在高速网络环境下NIO读写频率不一致的问题

易用

完善的JavaDoc，用户指南和样例

简洁简单

仅信赖于JDK1.5

看例子吧！

Server端：

**Java代码  [IMG_256](http://lippeng.iteye.com/blog/javascript:void())**

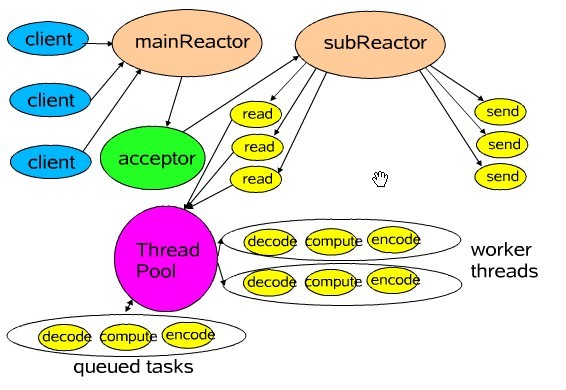
1. **package** me.hello.netty;
3. **import** org.jboss.netty.bootstrap.ServerBootstrap;
4. **import** org.jboss.netty.channel.\*;
5. **import** org.jboss.netty.channel.socket.nio.NioServerSocketChannelFactory;
6. **import** org.jboss.netty.handler.codec.string.StringDecoder;
7. **import** org.jboss.netty.handler.codec.string.StringEncoder;
9. **import** java.net.InetSocketAddress;
10. **import** java.util.concurrent.Executors;
12. /\*\*
13. \* God Bless You!
14. \* Author: Fangniude
15. \* Date: 2013-07-15
16. \*/
17. **public** **class** NettyServer {
18. **public** **static** **void** main(String[] args) {
19. ServerBootstrap bootstrap = **new** ServerBootstrap(**new** NioServerSocketChannelFactory(Executors.newCachedThreadPool(), Executors.newCachedThreadPool()));
21. // Set up the default event pipeline.
22. bootstrap.setPipelineFactory(**new** ChannelPipelineFactory() {
23. @Override
24. **public** ChannelPipeline getPipeline() **throws** Exception {
25. **return** Channels.pipeline(**new** StringDecoder(), **new** StringEncoder(), **new** ServerHandler());
26. }
27. });
29. // Bind and start to accept incoming connections.
30. Channel bind = bootstrap.bind(**new** InetSocketAddress(8000));
31. System.out.println("Server已经启动，监听端口: " + bind.getLocalAddress() + "， 等待客户端注册。。。");
32. }
34. **private** **static** **class** ServerHandler **extends** SimpleChannelHandler {
35. @Override
36. **public** **void** messageReceived(ChannelHandlerContext ctx, MessageEvent e) **throws** Exception {
37. **if** (e.getMessage() **instanceof** String) {
38. String message = (String) e.getMessage();
39. System.out.println("Client发来:" + message);
41. e.getChannel().write("Server已收到刚发送的:" + message);
43. System.out.println("\n等待客户端输入。。。");
44. }
46. **super**.messageReceived(ctx, e);
47. }
49. @Override
50. **public** **void** exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e) **throws** Exception {
51. **super**.exceptionCaught(ctx, e);
52. }
54. @Override
55. **public** **void** channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) **throws** Exception {
56. System.out.println("有一个客户端注册上来了。。。");
57. System.out.println("Client:" + e.getChannel().getRemoteAddress());
58. System.out.println("Server:" + e.getChannel().getLocalAddress());
59. System.out.println("\n等待客户端输入。。。");
60. **super**.channelConnected(ctx, e);
61. }
62. }
63. }

客户端：

**Java代码  [IMG_257](http://lippeng.iteye.com/blog/javascript:void())**

1. **package** me.hello.netty;
3. **import** org.jboss.netty.bootstrap.ClientBootstrap;
4. **import** org.jboss.netty.channel.\*;
5. **import** org.jboss.netty.channel.socket.nio.NioClientSocketChannelFactory;
6. **import** org.jboss.netty.handler.codec.string.StringDecoder;
7. **import** org.jboss.netty.handler.codec.string.StringEncoder;
9. **import** java.io.BufferedReader;
10. **import** java.io.InputStreamReader;
11. **import** java.net.InetSocketAddress;
12. **import** java.util.concurrent.Executors;
14. /\*\*
15. \* God Bless You!
16. \* Author: Fangniude
17. \* Date: 2013-07-15
18. \*/
19. **public** **class** NettyClient {
21. **public** **static** **void** main(String[] args) {
22. // Configure the client.
23. ClientBootstrap bootstrap = **new** ClientBootstrap(**new** NioClientSocketChannelFactory(Executors.newCachedThreadPool(), Executors.newCachedThreadPool()));
25. // Set up the default event pipeline.
26. bootstrap.setPipelineFactory(**new** ChannelPipelineFactory() {
27. @Override
28. **public** ChannelPipeline getPipeline() **throws** Exception {
29. **return** Channels.pipeline(**new** StringDecoder(), **new** StringEncoder(), **new** ClientHandler());
30. }
31. });
33. // Start the connection attempt.
34. ChannelFuture future = bootstrap.connect(**new** InetSocketAddress("localhost", 8000));
36. // Wait until the connection is closed or the connection attempt fails.
37. future.getChannel().getCloseFuture().awaitUninterruptibly();
39. // Shut down thread pools to exit.
40. bootstrap.releaseExternalResources();
41. }
43. **private** **static** **class** ClientHandler **extends** SimpleChannelHandler {
44. **private** BufferedReader sin = **new** BufferedReader(**new** InputStreamReader(System.in));
46. @Override
47. **public** **void** messageReceived(ChannelHandlerContext ctx, MessageEvent e) **throws** Exception {
48. **if** (e.getMessage() **instanceof** String) {
49. String message = (String) e.getMessage();
50. System.out.println(message);
52. e.getChannel().write(sin.readLine());
54. System.out.println("\n等待客户端输入。。。");
55. }
57. **super**.messageReceived(ctx, e);
58. }
60. @Override
61. **public** **void** channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) **throws** Exception {
62. System.out.println("已经与Server建立连接。。。。");
63. System.out.println("\n请输入要发送的信息：");
64. **super**.channelConnected(ctx, e);
66. e.getChannel().write(sin.readLine());
67. }
68. }
69. }

Netty整体架构



Netty组件

ChannelFactory

Boss

Worker

Channel

ChannelEvent

Pipeline

ChannelContext

Handler

Sink

Server端核心类

NioServerSocketChannelFactory

NioServerBossPool

NioWorkerPool

NioServerBoss

NioWorker

NioServerSocketChannel

NioAcceptedSocketChannel

DefaultChannelPipeline

NioServerSocketPipelineSink

Channels

ChannelFactory

Channel工厂，很重要的类

保存启动的相关参数

NioServerSocketChannelFactory

NioClientSocketChannelFactory

NioDatagramChannelFactory

这是Nio的，还有Oio和Local的

SelectorPool

Selector的线程池

NioServerBossPool 默认线程数：1

NioClientBossPool      1

NioWorkerPool      2 \* Processor

NioDatagramWorkerPool

Selector

选择器，很核心的组件

NioServerBoss

NioClientBoss

NioWorker

NioDatagramWorker

Channel

通道

NioServerSocketChannel

NioClientSocketChannel

NioAcceptedSocketChannel

NioDatagramChannel

Sink

负责和底层的交互

如bind，Write，Close等

NioServerSocketPipelineSink

NioClientSocketPipelineSink

NioDatagramPipelineSink

Pipeline

负责维护所有的Handler

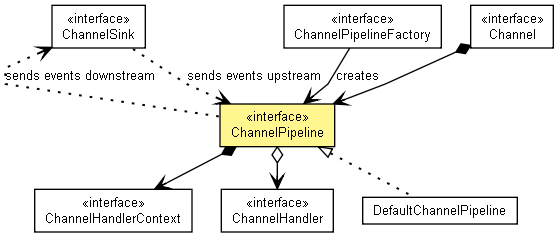
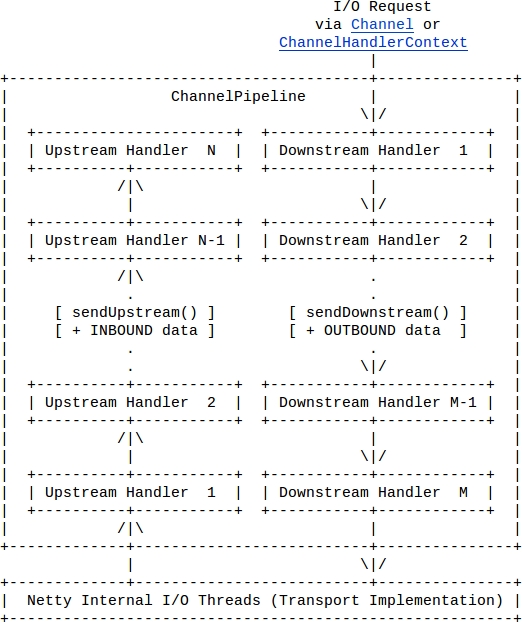
ChannelContext

一个Channel一个，是Handler和Pipeline的中间件

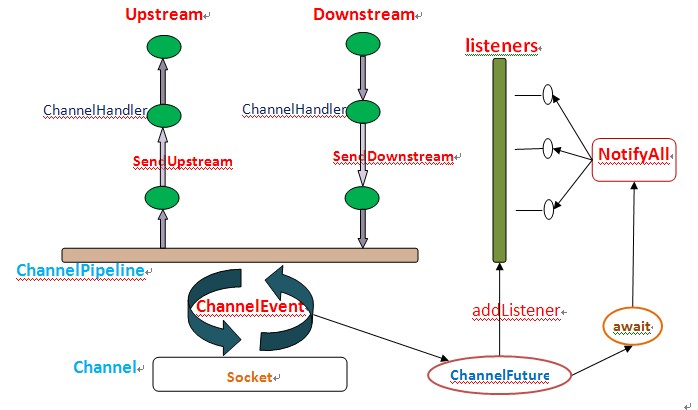
Handler

对Channel事件的处理器

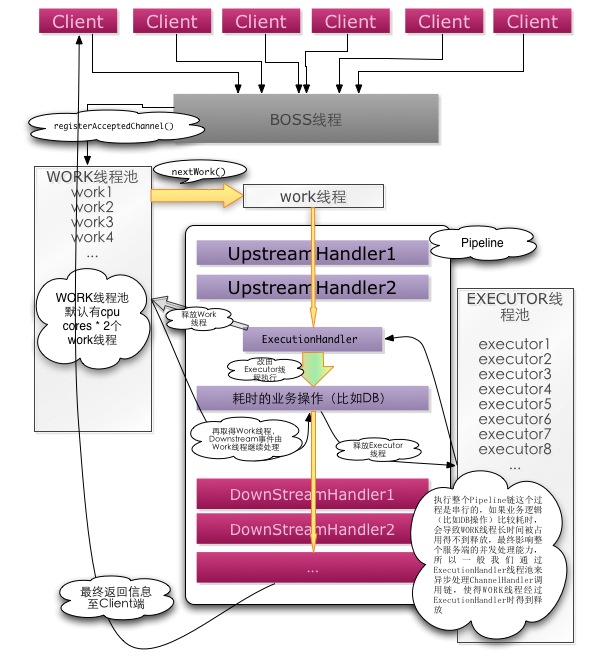
ChannelPipeline

优秀的设计----事件驱动



优秀的设计----线程模型



注意事项

解码时的Position

Channel的关闭

更多Handler

Channel的关闭

用完的Channel，可以直接关闭；

1、ChannelFuture加Listener

2、writeComplete

一段时间没用，也可以关闭

TimeoutHandler

# 机器学习

推荐完全没有开始学习的同学，可以先去集智（[https://jizhi.im](https://link.zhihu.com/?target=https://jizhi.im)）的社区看看,看看他们都在解决什么问题，对**机器学习**和深度学习有一个初步的印象，这对你看以下的若干段落会比较有帮助，因为有些词汇比较生涩。如果你已经有python基础，可以选择直接从《**机器学习**实战》开始你的**机器学习**之旅。如果你不知道如何入手，这里或许能提供一些入门的方法：[大一计算机学生如何学习**机器学习**？ - 知乎](https://www.zhihu.com/question/58990366/answer/162145071)。

**概览：**

**机器学习**是计算机科学的一个子领域，1959年一个叫Arthur Samuel的人给出了**机器学习**的定义“计算机有能力去学习，而不是通过预先准确实现的代码”。在人工智能领域，**机器学习**逐渐发展成模式识别和计算科学理论的研究。**机器学习**探索可以预测数据的算法的研究和建设--比如有些算法克服了严谨的静态的程序流程，通过从简单的数据中构建简单的模型来进行数据预测或者下决定。**机器学习**被应用于大规模计算性任务，尤其是当设计和编程过程中拥有良好表现的确切算法是难以实现或者不能实现的时候。简单的几个应用：email过滤，检测网络入侵者，OCR，排序学习，计算机视觉。

**机器学习**是非常接近（重叠）计算统计学的，关注于通过计算机来实现预测-决定这一过程。它与数学最优化有很强的关联性（提供方法，理论，应用领域）。**机器学习**有时候可以跟数据挖掘混为一谈，后者关注更多的是探索性的数据统计并且因为无监督学习而出名。**机器学习**同样可以是无监督的，并且可被用来学习和建立实体行为上的基准线，并且被用来发现有意义的异常。

在数据统计领域，**机器学习**是一种用来设计复杂模型和算法并以此实现预测功能的方法。在商业领域，**机器学习**因为预测分析而闻名。这些统计模型允许研究者，数据科学家，工程师和统计学家去“产出可信赖的，可重复的决策和结果”，并且通过对数据的历史关系和趋势的学习发现“隐藏的视角”。

从2016年起，**机器学习**成了流行词，并且通过Gartner 在2016年的大肆宣传，**机器学习**到达了不合理的期望的峰顶。有效的**机器学习**是困难的，因为找到模式是困难的并且经常出现训练数据不够的情况。导致**机器学习**程序经常不能交付结果。

Tom M. Mitchell 提出了一种被广泛引用的，更正式的定义：对于某类任务T和性能度量P，如果一个计算机程序在T上以P衡量的性能随着经验E而自我完善，那么我们称这个计算机程序在从经验E中学习。这个定义对**机器学习**在基础的操作上而不是认知术语上来说是重要的，因此根据Alan Turing在他的paper----“Computing Machinery and Intelligence”中的提议，问题“机器可以思考么？”可以被替换为“机器可以做我们可以做的么？”。在这个假说中，他想要实现构建一台思考的机器可能着了魔。

**不同的类型和任务**

**机器学习**任务往往可以被划分为三种大类，取决于可从训练系统获得的信号量或者反馈的属性。

这些分别是：

* 监督学习：计算机获得简单的输入给出期望的输出，过程是通过一个“训练模型”，学习通用的准则来从输入映射到输出。
* 无监督学习：没有给出标记用来学习算法，让它自己去发现输入的结构。无监督学习自己可以被当成一个目标或者一个实现结果的途径（特征学习）。
* 增强学习：一个计算机程序与动态环境交互，同时表现出确切目标（比如驾驶一辆交通工具或者玩一个游戏对抗一个对手）。这个程序的奖惩机制会作为反馈，实现它在问题领域中的导航。

处于监督学习和无监督学习之间的是半监督学习，“teacher”给出一个不完整的训练信号：一个训练集丢失了部分目标输出。传导是原则中的一个特殊的例子，除了那部分丢失的目标，整个问题实例的集合在学习过程中是被知晓的。

在别的**机器学习**问题的分类中，学会学习这一过程需要靠学习它自己，这归纳偏向于之前的经验。

发展性学习则阐明了机器人学习，生成它自己学习情况的序列，通过不断地自我探索、与人类“教师”的不断社会性交互、使用导向性的机制（比如主动学习，熟化，机器协同，模仿这一过程），进而去累积地获得各部分的独特技巧。

当一个人开始考虑获得一个**机器学习**过的系统的输出，另一类**机器学习**的任务就出现了：

* 从分类上来说，输出被分为两个或者多个类别，并且学习者必须制造出一个模型用来分配目前没有出现的输入数据提供给一个或者更多的类别。这是传统解决方法，使用监督学习。垃圾邮件过滤器就是一个典型的分类，输入是email信息并且类别分为垃圾邮件和非垃圾邮件。
* 从回归上来说，同样是一个监督学习的问题，输出是连续的而不是离散的。
* 从聚类上来说，一个输入集应该被分为若干个组。不同于分类的做法，这些组之前并不被预先知道，这使其被当做无监督学习的任务来处理。
* 密度估计可以在某些地方找到不同版本的输入。
* 降维处理简化了输入，通过映射它们给更低维度的空间。主题模拟是一种近似的问题解决办法，尤其是当一个程序被给予了一个序列的人类语言的文档，并且任务是找出哪些文档的主题接近时。

**历史渊源以及和别的领域的关系：**

作为一项科学尝试，**机器学习**在人工智能领域不断地探索，现在已经长大。在人工智能的早期，作为一门学术学科，许多研究者对机器从数据中学习非常感兴趣。他们尝试着用多种抽象的手段去解决问题，比如术语----“神经网络”。这些感知机一样的玩意儿和一些其他的模型被重新改变为统计学上的广义线性模型。概率推理也被应用于此，尤其是自动化的医疗诊断系统。

但是，“过于强调逻辑，以已有知识为基础的方法”这一现象，造成了人工智能和**机器学习**的分歧与割裂。概率系统被数据的获得和表现，这两者的理论和实践中出现的问题所困扰。到1980年，专家系统开始主宰人工智能，并且统计学开始失宠。致力于抽象的/以现有知识为基础的的学习继续影响着人工智能，引领了归纳逻辑编程这一潮流，但是更多的统计学上的线性研究开始完全地跌出了人工智能，尤其是在模式识别和信息检索领域。神经网络研究此时被人工智能和计算机科学同时放弃。与此相关的整个系列，也因为 Hopfield, Rumelhart and Hinton三者在20世纪80年代中期对反向传播算法的成功革新，被人工智能/计算机科学领域当成一种联结机制而排除。

**机器学习**，被改组成了一个独立的领域，在20世纪90年代开始走向繁荣。这个领域改变了它的目标，从实现人工智能到解决实际问题。它的关注点从继承自人工智能的象征性做法转换为从统计学和概率学上借来的方法和模型。它同时也从英特网不断增长的数字化信息、分发有效性等方面获利。

**机器学习**和数据挖掘经常使用同样的方法并具有明显地重叠，但是不同之处是**机器学习**关注于预测，是从训练数据中学习，基于已知的特性进行预测。数据挖掘关注于从数据中发现未知的特性。数据挖掘使用和很多**机器学习**的手法，但是持有的是不同的目标。从另一方面说，**机器学习**同样使用了数据挖掘的一些做法，比如“非监督学习”或者预处理的步骤去提高学习器的准确性。大量的困惑出现在两个研究团体之间，这来自于两个团体的基础假设：在**机器学习**领域，表现是通常用来衡量再次产生已知知识的能力，但是在知识探索和数据挖掘(KDD)领域，关键任务是发现先前不知道的知识。衡量已知的知识，一套监督学习的处理方法将会比无知的做法（无监督学习）容易呈现出更好的性能，虽然在典型的KDD任务中，监督学习不能被使用，因为训练用的数据的无用性。

**机器学习**同时与优化有着亲密的关系：许多学习问题可以被作为公式，用来使训练样例集中损失函数最小化。损失函数在预测被训练的模型和实际问题实例上表现出了差异性（举例来说，就分类来说，一个想要程序想要分派标签给一个实例，并且模型被训练为正确预测，预先指派标签给样例数据集）。两个领域的不同之处源自目标的广义化：一个优化算法关注于在训练集上可以被最小化损失，而**机器学习**是关心最小化损失在看不到的例子上。

**机器学习与统计学的关系**

**机器学习**和统计学是非常接近的两个领域。根据 Michael I. Jordan在**机器学习**领域的理念，从方法论原则到理论工具，在统计学领域是有一段很长的史前史。他也建议数据科学这一术语作为全部领域的前置。

Leo Breiman区别两个统计学的模型：数据模型和算法模型，在算法模型中意味着或多或少包含着**机器学习**的算法，比如随机森林（Random forest）。

一些统计学家已经采纳了**机器学习**中的一些做法，引申出了一个联结领域-----统计学习。

**机器学习理论**

一个学习者的核心目标就是从经验中形成概念（推广）。概括是学习机器通过在学习一个数据集后在新的、未知的例子/任务中根据上下文精确表现的一种能力。这些训练例子来自一些通用的不知道可能性的分发版本，并且学习者不得不去构建一个通用模型来使它在新的例子中产生足够准确预测。

**机器学习**算法中的计算分析和它们的表现是一个理论计算机科学的分支，其中理论计算科学因计算学习理论而出名。因为训练集是有限的，并且将来是不确定的，学习理论通常并不做出对算法表现的保证。反而，在表现上出现两极间概率性的抖动是很常见的。偏置方差分解是一个量化概括过程出现的错误的方法。

为了在泛化过程环境中的最好表现，假说的复杂性应该匹配数据下函数的复杂性。如果假说的复杂性小于函数，那么模型会对数据欠拟合。如果模型的复杂性作为回应而增强，那么训练中的错误会减少。但是如果假说过于复杂，那么模型会过拟合，并且泛化性会变差。

除了性能界限之外，计算学习理论家研究了时间的复杂度和学习的可能性。在计算学习理论中，一个计算被认为是可行的，如果它可以在多项式时间内被执行完毕。这里有两种时间复杂度结果。积极结果展现出一个确定类别的函数，它可以在多项式时间内被学习。消极的结果展现出确定类别不能在多项式时间内被学习。

**机器学习方法**

决策树学习：决策树学习使用了一个决策树作为预测性模型，映射一个对象的观察结果给其目标价值一个推论。

关联规则学习：关联规则学习是一种用来在大型数据库中发现变量之间的有趣联系的方法。

人工神经网络：一个人工神经网络学习（ANN）算法，通常被称为神经网络（NN），是一个由生物的神经网络所激发出的一个算法。计算结构是由联结的人工神经元组所构成，通过联结式的方法来传递信息和计算。现代神经网络是非线性的统计学数据模型工具。它们通常被用来在输入和输出之间模拟复杂关系，找到数据中的关系，或者在观测变量中从不知道的节点捕获统计学结构。

深度学习：个人不能承受硬件的价格和GPU的发展推动了这些年深度学习的进步，深度学习是由人工神经网络中的多个隐藏层组成的。这条道路试图去模拟人脑的过程，光、声进入视觉和听觉。一些成功的应用有计算机视觉和演讲识别。

归纳逻辑编程：归纳逻辑编程（ILP）是一门用逻辑编程控制规则的学科，它使用统一的表示法来处理输入样例，背景知识和假说。给定已知的背景知识的编码和一组被表示为事实的逻辑数据库的示例，ILP系统将派生出一个假设的逻辑程序，该程序包含所有积极的和没有负面的示例。归纳编程是一个相关的领域，它考虑任何一种表示假设(而不仅仅是逻辑编程)的编程语言，例如函数式编程。

支持向量机：支持向量机是一系列关于监督学习在分类和回归上的应用。给出训练样本的数据集，每一个标记属于两类中的一类，一个SVM训练算法构成了一个模型，可以用来预测一个新的样本是否进入一个类别或者是另一个。

集群：集群分析是将一组观察结果分配到子集(称为集群)，这样，同一集群中的观察与一些预先确定的标准或标准相似，而来自不同集群的观察则不同。不同的聚类技术对数据的结构作出不同的假设，通常由一些相似性度量定义，并通过内部紧度(相同集群的成员之间的相似性)和不同的集群之间的分离来评估。其他方法基于估计的密度和图连通性。摘要聚类是一种非引导性学习的方法，是一种统计数据分析的常用技术。

贝叶斯网络：一个贝叶斯网络，信任网络或者有向无环图模型是一个概率性图的模型，它通过有向无环图代表了一系列的随机变量和他们的条件独立性。举例，一个贝叶斯网络代表着疾病和症状可能的关系。给出症状，网络可以被用来计算疾病出现的可能性。有效的算法存在于执行推理和学习的过程中。

增强学习：增强学习关心代理人如何在一个环境中采取行动，从而最大化一些长期受益的概念。增强学习算法尝试去寻找一些策略，映射当前世界的状态给代理在这些状态中应该采取的行动。

特征学习：一些学习算法，主要是无监督的学习算法，旨在发现训练中提供的输入的更好的表示。经典的例子包括主成分分析和聚类分析。特征学习算法常常试图保存他们的信息输入同时变换方式使它有用,在执行分类或预测之前经常作为预处理步骤,允许重建输入来自未知的数据生成版本,而不一定是必须可靠的设置。流行学习算法试图在学习表现是低维度的约束条件下实现这一目标。稀疏编码算法试图在学习表示稀疏(有很多个0)的约束条件下这样做。多线性子空间学习算法的目标是直接从多维数据的张量表示中学习低维表示，而不需要将它们重新组合成(高维的)向量。深度学习算法发现了多个层次的表示形式，或者说是一种层次结构，在(或生成)低层次特性的基础上定义了更高级、更抽象的特性。有人认为，智能机器是一种可以学习一种表达方式，它可以分解解释所观察到的数据的各种变化的潜在因素。

相似度量学习：在这个问题中，学习机被给予了很多对相似或者不相似的例子。它需要去学习一个相似的函数，以用来预测一个新的对象是否相似。它有时被用到推荐系统中。

稀疏字典学习：在这种方法中，一个基准面被表示为基函数的线性组合，那么这些系数被认为是稀疏的。让x是一个D维的基准，D是一个d乘以n的矩阵，每一列D代表一个基函数。r是用d表示x的系数。从数学上讲，稀疏字典学习意味着解决

这个问题，这里r是稀疏的。一般来说，n被假定为大于d，以允许稀疏表示的自由。学习字典和稀疏表示是非常困难NP问题，很难解决。一种流行的启发式学习方法是k-svd。稀疏字典学习被应用在很多场景。在分类中，问题是确定一个从未被发现的数据属于哪个类。假设每个类的字典已经构建好了。然后，一个新的数据与类相关联，这样就可以用相应的字典来表示最稀疏的数据了。稀疏字典学习也被应用于图像去噪。关键的地方是，一个清晰的图像补丁可以用图像字典来表示，但是噪音是不能的。

遗传算法：遗传算法是一种启发式搜索，它模仿自然选择的过程，并且使用一些突变和变向来生成新的基因型，以找到好的情况解决问题。在**机器学习**中，遗传算法在20世纪80年代和90年代使用过。反之，**机器学习**技术被用来提高遗传和进化算法的表现。

基于规则的**机器学习**：基于规则的**机器学习**是任何**机器学习**方法的通用术语，它可以识别、学习或发展规则来存储、操作或应用知识。基于规则的**机器学习**者的定义特征是一组关系规则的标识和利用，这些规则集合了系统所捕获的知识。这与其他**机器学习**者形成鲜明对比，他们通常会识别出一种特殊的模型，这种模型可以普遍应用于任何实例，以便做出预测。基于规则的**机器学习**方法包括学习分类器系统、关联规则学习和人工免疫系统。

学习分类器系统：学习分类器系统（LCS）是一个基于规则**机器学习**算法的家族，它每次结合一个发现组件和一个学习组件。它们试图鉴别一套上下文相关的规则，用分段的方式共同地存储和应用知识，以达到预测的目的。

**应用：**

* 自适应网站
* 情感计算
* 生物信息学
* 脑机接口
* 化学信息学
* DNA序列分类
* 计算解剖学
* 计算机视觉，包括物体识别
* 信用卡欺诈检测
* 博弈
* 信息检索
* 网络欺诈检测
* 市场营销
* **机器学习**控制
* 机器感知
* 医疗诊断
* 经济学
* 自然语言处理
* 自然语言理解
* 优化和启发式算法
* 在线广告
* 推荐系统
* 机器人移动
* 搜索引擎
* 情绪分析
* 序列挖掘
* 软件工程
* 演讲和手写识别
* 金融市场分析
* 健康监测
* 句法模式识别
* 用户行为分析
* 翻译

# MongoDB、Cassandra 和 HBase 三种 NoSQL 数据库比较

2015/09/24 · [IT技术](http://blog.jobbole.com/category/it-tech/) · [1 评论](http://blog.jobbole.com/91923/#article-comment)· [Cassandra](http://blog.jobbole.com/tag/cassandra/), [HBase](http://blog.jobbole.com/tag/hbase/), [MongoDB](http://blog.jobbole.com/tag/mongodb/), [NoSQL](http://blog.jobbole.com/tag/nosql/), [数据库](http://blog.jobbole.com/tag/database/)

分享到：**23**

本文由 [伯乐在线](http://blog.jobbole.com/) - [徐汉彬](http://www.jobbole.com/members/xhb7636553) 翻译。未经许可，禁止转载！  
英文出处：[infoworld](http://hansionxu.blog.163.com/blog/static/24169810920141021111650462/)。欢迎加入[翻译组](http://group.jobbole.com/category/feedback/trans-team/)。

Hadoop获得了许多大数据应用的信誉，但实际情况是，NoSQL数据库是一直处于更加广泛部署和更广泛的发展中的技术。尽管选择Hadoop作为应用存储，相对来说更直接简单。但是，具体采用什么样的NoSQL数据库是个值得思考的问题，毕竟，还有超过100种的NoSQL数据库。

我们应该选择哪一种？

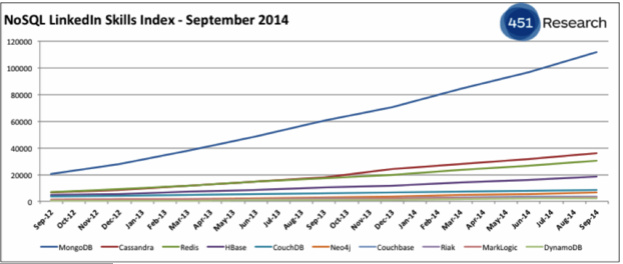
### 选择倾向

“任何像样规模的企业都会使用各种不同类型的数据存储技术，为应对各种不同类型的数据。”Martin Fowler认为，现实的情况是你没有足够的精力去学习更多的存储技术。

幸运的是，选择越来越容易，因为市场主要围绕在三个NoSQL数据库上：MongoDB，Cassandra（主要由DataStax开发的，诞生于Facebook），和HBase的（和Hadoop紧密关联在一起，也被相同社区开发出来）。

补充一点，我故意排除Redis。相对于大数据存储，它主要用于高速内存缓存数据应用。

从LinkedIn的451研究数据显示，市场上最具引力的是MongoDB、Cassandra和HBase：



这是LinkedIn的个人资料数据。我们认为是数据存储引擎，它通过收集工作、搜索等数据，来了解数据库的热门程度。而Oracle，SQL Server和MySQL的占据了统治地位，MongoDB的（第5位），Cassandra（第9位），和HBase的（第15位）。

为了更好解释为什么这三个数据库技术的如此耀眼，我问的每一个具有代表性的人，以确定它们成功关键因素：Kelly Stirman，MongoDB的产品总监;Patrick McFadin，DataStax的Cassandra首席布道师;和Justin Kestelyn，Cloudera高级总监。

但首先，我们需要了解为什么使用NoSQL的原因。

### 世界由非结构化数据构成

我们生活在一个数据越来越丰富的世界里，但是这些数据都不能整齐的展示在一个RDBMS（Relational Database Management System，关系数据库管理系）的行和列中。移动、社交和云计算催生了庞大的海量的数据。根据估计，世界上90％的数据是在过去两年中被创造，以及80％的商业数据是非结构化的。更重要的是，非结构化数据的增长速度是结构化数据的两倍。

随着世界的变化，数据管理要求开始超越传统的关系型数据库的有效范围。最早关注这个问题解决方案的机构，包括Web技术的先驱、政府机构、从事信息技术服务的公司。

现在越来越多，形形色色的公司都希望利用类似的NoSQL和Hadoop作为替代品：通过NoSQL来建立业务运营应用，以及Hadoop来创建数据挖掘的应用程序，来帮助公司对商业数据提供有力的研究。

### MongoDB：源于开发人员，为开发人员服务

在众多NoSQL的方案中，MongoDB的Stirman指出，MongoDB的瞄准了适合各种应用的平衡的方法。它的功能接近于传统的关系型数据库，MongoDB的用户不仅可以利用其横向扩展机器的云基础架构的优势，并且，因为它能够轻松定义各种灵活的数据模型，所以可以支持不同类型的数据集存储。

MongoDB通常是开发人员第一个尝试的NoSQL数据库，因为它是很容易学习。Will Shulman，MongoLab（一个MongoDB服务提供商）的CEO，是这样说的：

MongoDB中的成功在很大程度上是因为它数据结构存储的创新，让我们更容易和更具表现力地定义我们应用程序中的数据模型。在通常开发和应用场景中，和原有数据库具有相同的基本数据模型是有极大好处的，因为它简化了应用程序开发的任务，另一方面，消除了复杂的数据格式代码转换层。

当然，像任何其他技术一样，MongoDB中都有其长处和短处。 MongoDB是专门为OLTP（On-Line Transaction Processing，联机事务处理系统）模式。如果您需要复杂的事务处理，它不是一个好的选择。然而，MongoDB的简单性使其成为一个优秀的存储。

（注：MongoDB以文档的形式存储数据，不支持事务和表连接。因此查询的编写、理解和优化都容易得多。）

### Cassandra：规模化安全运行

三种数据库中，至少两种数据库具有简单特性：开发简单，操作简便。而MongoDB赢得人心的原因是简单的开发应用，Cassandra赢得人心是因为易于管理的规模。

DataStax的McFadin告诉我，用户往往倾向于使用Cassandra ，是因为特别在大规模集群下，增强一个关系型数据的性能、可靠性是非常困难的。一位前甲骨文DBA，McFadin是兴高采烈地发现，“复制和可扩放性是基础”，Cassandra 特点是从一开始设计就解决这个问题。

在RDBMS中的世界，数据库功能，拓展和复制对很多开发者用户来说，是一个难题。这个问题在过往的企业规模小的时候，不是一个大问题。而在今天，它很迅速地成为大问题。

我从McFadin和其他人那里获知，Cassandra在机器拓展部署上，表现特别出色。Cassandra自带的备份机制，保证各个数据中心的数据安全。至于增加容量到集群，“你只需启动一台新机器，并告诉Cassandra那里的新节点，”McFadin说，“然后，它完成其他剩下的事情。”

优秀的可拓展性，加上出色的写入和可观的查询性能，加起来成为Cassandra高性能的核心。

NoSQL的一篇文章认为Cassandra在集群规模管理方面非常出色，但它需要一个博士学位才能上手。事实并非如此，McFadin坚持认为：  
在复制、读取和写入是故意简单。你可以在几个小时内学会Cassandra的核心功能。在部署这项新技术的时候，为给开发者带来很多的信心，因为比较少引入“黑盒子”内的技术细节和复杂的故障模式原理。

这意味着主要的开发成本，是对Cassandra数据模型的理解，以及如何结合您的应用程序。鉴于Cassandra的CQL查询语言（类似于SQL，实际上不是SQL），McFadin说，学习这个也不困难。

更重要的是，他告诉我，“Cassandra回报给你的是，在一个数据库中：没有戏剧性的场景（故障）出现。这就是用户喜欢使用Cassandra的原因。”

### HBase：Hadoop的知心伙伴

HBase，像Cassandra一样是个通过key-value面向列存储的服务。因为它和Hadoop有着“共同血统”，被广泛使用。事实上，正如Cloudera的Kestelyn所说的那样，“HBase提供了一个基于记录的存储层，能够快速随机读取和写入数据，正好弥补了Hadoop的缺陷，Hadoop侧重系统吞吐量，而牺牲I / O读取效率为代价。”

Kestelyn接着说：  
更改有效录入到内存中，以达到最大的访问量，同时将数据保存到HDFS。这种设计使基于Hadoop的EDH（enterprise data hub，企业数据中心）服务，能够实时完成随机读写存储数据，但仍拥有HDFS的高容错性和耐用性。

Hadoop的亲和力，不是HBase数据库中的人气排名不断上升的唯一原因。类似Cassandra，HBase是Google的Bigtable的开源实现转化成的数据库，天然被设计为高可扩展性。

Hbase可以利用任何数量服务器的磁盘、内存和CPU资源，同时拥有极佳的扩展功能，如自动分片。当系统负载和性能要求不断增加，HBase的可通过简单增加服务器节点的方式无限拓展。 HBase从底层设计上保证，在确保数据一致性的同时，提供最佳性能。

但规模不是它的唯一用途。Kestelyn指出，“由于它与Hadoop的生态系统紧密集成，对于用户和应用程序来说，数据是容易获取的，可以通过SQL的方式查询（使用Cloudera的Impala，Phoenix，或Hive），甚至自由文本搜索（使用Cloudera Search）。“因此，HBase为开发人员提供了一种方法，利用现有通用的SQL语言，来建立在一个更成熟的分布式数据库。

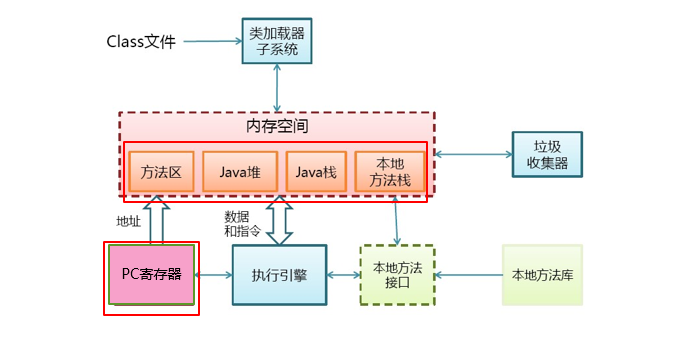
每种数据库技术都有自己的长处和不足，但这里评论的三种数据库，在大数据技术领域，占据了重要的位置。虽然未来可能还有一种全新的NoSQL数据库技术会挑战它们前三的位置，但目前的现实是，许多开发人员以及一批强大的成熟企业已经做出了它们的选择：MongoDB、Cassandra 和 HBase。

# [JVM内核解读](http://www.cnblogs.com/sz-zzm/p/5229405.html)

JVM是Java Virtual Machine（Java虚拟机）的缩写，这里要分清的是，JVM与java语言是相对独立的，也就是说只要有符合java语言规范的虚拟机也可以运行java语言。

1.JVM的启动流程

2.JVM基本结构



PC寄存器：

1.每个线程拥有一个PC寄存器

2.在线程创建时创建

3.指向下一条指令的地址

4.执行本地方法时，PC的值为undefined

方法区：

1.保存装载的类信息

　　类型的常量池

　　字段，方法信息

　　方法字节码

2.通常和永久区（perm）关联在一起

jdk1.6 String等常量信息置放于方法区

jdk1.7 移动到堆

java堆：

1.和程序开发密切相关

2.保存应用系统对象

3.所有线程共享

4.对分代GC（垃圾回收）来说，堆也是分代的

5.GC主要工作区间



java栈：

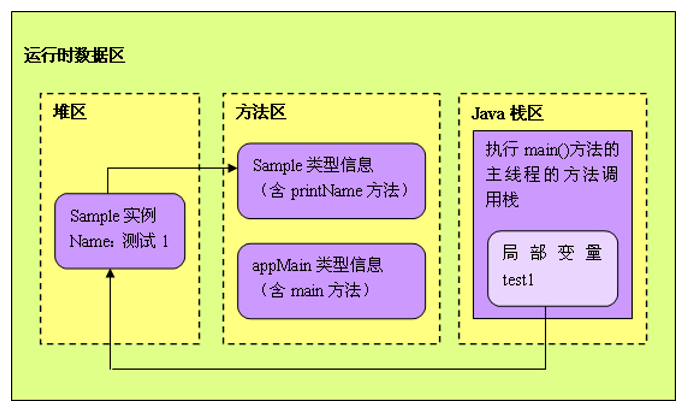
1.线程私有

2.由一系列帧组成（因此也称作帧栈）

3.帧保存一个方法的局部变量，操作数栈，常量池指针

4.每一个方法调用创建一个帧，并压栈

 3.堆栈方法交互



# 基于Dubbo框架构建分布式服务

[2015-03-31 22:30:45](http://shiyanjun.cn/archives/1075.html)    [Yanjun](http://shiyanjun.cn/archives/author/yanjun)

Dubbo是Alibaba开源的分布式服务框架，我们可以非常容易地通过Dubbo来构建分布式服务，并根据自己实际业务应用场景来选择合适的集群容错模式，这个对于很多应用都是迫切希望的，只需要通过简单的配置就能够实现分布式服务调用，也就是说服务提供方（Provider）发布的服务可以天然就是集群服务，比如，在实时性要求很高的应用场景下，可能希望来自消费方（Consumer）的调用响应时间最短，只需要选择Dubbo的Forking Cluster模式配置，就可以对一个调用请求并行发送到多台对等的提供方（Provider）服务所在的节点上，只选择最快一个返回响应的，然后将调用结果返回给服务消费方（Consumer），显然这种方式是以冗余服务为基础的，需要消耗更多的资源，但是能够满足高实时应用的需求。  
有关Dubbo服务框架的简单使用，可以参考我的其他两篇文章（《基于Dubbo的Hessian协议实现远程调用》，《Dubbo实现RPC调用使用入门》，后面参考链接中已给出链接），这里主要围绕Dubbo分布式服务相关配置的使用来说明与实践。

**Dubbo服务集群容错**

假设我们使用的是单机模式的Dubbo服务，如果在服务提供方（Provider）发布服务以后，服务消费方（Consumer）发出一次调用请求，恰好这次由于网络问题调用失败，那么我们可以配置服务消费方重试策略，可能消费方第二次重试调用是成功的（重试策略只需要配置即可，重试过程是透明的）；但是，如果服务提供方发布服务所在的节点发生故障，那么消费方再怎么重试调用都是失败的，所以我们需要采用集群容错模式，这样如果单个服务节点因故障无法提供服务，还可以根据配置的集群容错模式，调用其他可用的服务节点，这就提高了服务的可用性。  
首先，根据Dubbo文档，我们引用文档提供的一个架构图以及各组件关系说明，如下所示：  
  
上述各个组件之间的关系（引自Dubbo文档）说明如下：

* 这里的Invoker是Provider的一个可调用Service的抽象，Invoker封装了Provider地址及Service接口信息。
* Directory代表多个Invoker，可以把它看成List，但与List不同的是，它的值可能是动态变化的，比如注册中心推送变更。
* Cluster将Directory中的多个Invoker伪装成一个Invoker，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个。
* Router负责从多个Invoker中按路由规则选出子集，比如读写分离，应用隔离等。
* LoadBalance负责从多个Invoker中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选。

我们也简单说明目前Dubbo支持的集群容错模式，每种模式适应特定的应用场景，可以根据实际需要进行选择。Dubbo内置支持如下6种集群模式：

* Failover Cluster模式

配置值为failover。这种模式是Dubbo集群容错默认的模式选择，调用失败时，会自动切换，重新尝试调用其他节点上可用的服务。对于一些幂等性操作可以使用该模式，如读操作，因为每次调用的副作用是相同的，所以可以选择自动切换并重试调用，对调用者完全透明。可以看到，如果重试调用必然会带来响应端的延迟，如果出现大量的重试调用，可能说明我们的服务提供方发布的服务有问题，如网络延迟严重、硬件设备需要升级、程序算法非常耗时，等等，这就需要仔细检测排查了。  
例如，可以这样显式指定Failover模式，或者不配置则默认开启Failover模式，配置示例如下：

|  |  |
| --- | --- |
| 1 | <**dubbo:service** interface="org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService" version="1.0.0" |

|  |  |
| --- | --- |
| 2 | cluster="failover" retries="2" timeout="100" ref="chatRoomOnlineUserCounterService" protocol="dubbo" > |

|  |  |
| --- | --- |
| 3 | <**dubbo:method** name="queryRoomUserCount" timeout="80" retries="2" /> |

|  |  |
| --- | --- |
| 4 | </**dubbo:service**> |

上述配置使用Failover Cluster模式，如果调用失败一次，可以再次重试2次调用，服务级别调用超时时间为100ms，调用方法queryRoomUserCount的超时时间为80ms，允许重试2次，最坏情况调用花费时间160ms。如果该服务接口org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService还有其他的方法可供调用，则其他方法没有显式配置则会继承使用dubbo:service配置的属性值。

* Failfast Cluster模式

配置值为failfast。这种模式称为快速失败模式，调用只执行一次，失败则立即报错。这种模式适用于非幂等性操作，每次调用的副作用是不同的，如写操作，比如交易系统我们要下订单，如果一次失败就应该让它失败，通常由服务消费方控制是否重新发起下订单操作请求（另一个新的订单）。

* Failsafe Cluster模式

配置值为failsafe。失败安全模式，如果调用失败， 则直接忽略失败的调用，而是要记录下失败的调用到日志文件，以便后续审计。

* Failback Cluster模式

配置值为failback。失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

* Forking Cluster模式

配置值为forking。并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。

* Broadcast Cluster模式

配置值为broadcast。广播调用所有提供者，逐个调用，任意一台报错则报错（2.1.0开始支持）。通常用于通知所有提供者更新缓存或日志等本地资源信息。  
上面的6种模式都可以应用于生产环境，我们可以根据实际应用场景选择合适的集群容错模式。如果我们觉得Dubbo内置提供的几种集群容错模式都不能满足应用需要，也可以定制实现自己的集群容错模式，因为Dubbo框架给我提供的扩展的接口，只需要实现接口com.alibaba.dubbo.rpc.cluster.Cluster即可，接口定义如下所示：

|  |  |
| --- | --- |
| 01 | @SPI(FailoverCluster.NAME) |

|  |  |
| --- | --- |
| 02 | **public** **interface** Cluster { |

|  |  |
| --- | --- |
| 03 |  |

|  |  |
| --- | --- |
| 04 | /\*\* |

|  |  |
| --- | --- |
| 05 | \* Merge the directory invokers to a virtual invoker. |

|  |  |
| --- | --- |
| 06 | \* @param <T> |

|  |  |
| --- | --- |
| 07 | \* @param directory |

|  |  |
| --- | --- |
| 08 | \* @return cluster invoker |

|  |  |
| --- | --- |
| 09 | \* @throws RpcException |

|  |  |
| --- | --- |
| 10 | \*/ |

|  |  |
| --- | --- |
| 11 | @Adaptive |

|  |  |
| --- | --- |
| 12 | <T> Invoker<T> join(Directory<T> directory) **throws** RpcException; |

|  |  |
| --- | --- |
| 13 |  |

|  |  |
| --- | --- |
| 14 | } |

关于如何实现一个自定义的集群容错模式，可以参考Dubbo源码中内置支持的汲取你容错模式的实现，6种模式对应的实现类如下所示：

|  |  |
| --- | --- |
| 1 | com.alibaba.dubbo.rpc.cluster.support.FailoverCluster |

|  |  |
| --- | --- |
| 2 | com.alibaba.dubbo.rpc.cluster.support.FailfastCluster |

|  |  |
| --- | --- |
| 3 | com.alibaba.dubbo.rpc.cluster.support.FailsafeCluster |

|  |  |
| --- | --- |
| 4 | com.alibaba.dubbo.rpc.cluster.support.FailbackCluster |

|  |  |
| --- | --- |
| 5 | com.alibaba.dubbo.rpc.cluster.support.ForkingCluster |

|  |  |
| --- | --- |
| 6 | com.alibaba.dubbo.rpc.cluster.support.AvailableCluster |

可能我们初次接触Dubbo时，不知道如何在实际开发过程中使用Dubbo的集群模式，后面我们会以Failover Cluster模式为例开发我们的分布式应用，再进行详细的介绍。

**Dubbo服务负载均衡**

Dubbo框架内置提供负载均衡的功能以及扩展接口，我们可以透明地扩展一个服务或服务集群，根据需要非常容易地增加/移除节点，提高服务的可伸缩性。Dubbo框架内置提供了4种负载均衡策略，如下所示：

* Random LoadBalance：随机策略，配置值为random。可以设置权重，有利于充分利用服务器的资源，高配的可以设置权重大一些，低配的可以稍微小一些
* RoundRobin LoadBalance：轮询策略，配置值为roundrobin。
* LeastActive LoadBalance：配置值为leastactive。根据请求调用的次数计数，处理请求更慢的节点会受到更少的请求
* ConsistentHash LoadBalance：一致性Hash策略，具体配置方法可以参考Dubbo文档。相同调用参数的请求会发送到同一个服务提供方节点上，如果某个节点发生故障无法提供服务，则会基于一致性Hash算法映射到虚拟节点上（其他服务提供方）

在实际使用中，只需要选择合适的负载均衡策略值，配置即可，下面是上述四种负载均衡策略配置的示例：

|  |  |
| --- | --- |
| 1 | <**dubbo:service** interface="org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService" version="1.0.0" |

|  |  |
| --- | --- |
| 2 | cluster="failover" retries="2" timeout="100" loadbalance="random" |

|  |  |
| --- | --- |
| 3 | ref="chatRoomOnlineUserCounterService" protocol="dubbo" > |

|  |  |
| --- | --- |
| 4 | <**dubbo:method** name="queryRoomUserCount" timeout="80" retries="2" loadbalance="leastactive" /> |

|  |  |
| --- | --- |
| 5 | </**dubbo:service**> |

上述配置，也体现了Dubbo配置的继承性特点，也就是dubbo:service元素配置了loadbalance=”random”，则该元素的子元素dubbo:method如果没有指定负载均衡策略，则默认为loadbalance=”random”，否则如果dubbo:method指定了loadbalance=”leastactive”，则使用子元素配置的负载均衡策略覆盖了父元素指定的策略（这里调用queryRoomUserCount方法使用leastactive负载均衡策略）。  
当然，Dubbo框架也提供了实现自定义负载均衡策略的接口，可以实现com.alibaba.dubbo.rpc.cluster.LoadBalance接口，接口定义如下所示：

|  |  |
| --- | --- |
| 01 | /\*\* |

|  |  |
| --- | --- |
| 02 | \* LoadBalance. (SPI, Singleton, ThreadSafe) |

|  |  |
| --- | --- |
| 03 | \* |

|  |  |
| --- | --- |
| 04 | \* <a href="<http://en.wikipedia.org/wiki/Load_balancing_>(computing)">Load-Balancing</a> |

|  |  |
| --- | --- |
| 05 | \* |

|  |  |
| --- | --- |
| 06 | \* @see com.alibaba.dubbo.rpc.cluster.Cluster#join(Directory) |

|  |  |
| --- | --- |
| 07 | \* @author qian.lei |

|  |  |
| --- | --- |
| 08 | \* @author william.liangf |

|  |  |
| --- | --- |
| 09 | \*/ |

|  |  |
| --- | --- |
| 10 | @SPI(RandomLoadBalance.NAME) |

|  |  |
| --- | --- |
| 11 | **public** **interface** LoadBalance { |

|  |  |
| --- | --- |
| 12 |  |

|  |  |
| --- | --- |
| 13 | /\*\* |

|  |  |
| --- | --- |
| 14 | \* select one invoker in list. |

|  |  |
| --- | --- |
| 15 | \* @param invokers invokers. |

|  |  |
| --- | --- |
| 16 | \* @param url refer url |

|  |  |
| --- | --- |
| 17 | \* @param invocation invocation. |

|  |  |
| --- | --- |
| 18 | \* @return selected invoker. |

|  |  |
| --- | --- |
| 19 | \*/ |

|  |  |
| --- | --- |
| 20 | @Adaptive("loadbalance") |

|  |  |
| --- | --- |
| 21 | <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) **throws** RpcException; |

|  |  |
| --- | --- |
| 22 |  |

|  |  |
| --- | --- |
| 23 | } |

如何实现一个自定义负载均衡策略，可以参考Dubbo框架内置的实现，如下所示的3个实现类：

|  |  |
| --- | --- |
| 1 | com.alibaba.dubbo.rpc.cluster.loadbalance.RandomLoadBalance |

|  |  |
| --- | --- |
| 2 | com.alibaba.dubbo.rpc.cluster.loadbalance.RoundRobinLoadBalance |

|  |  |
| --- | --- |
| 3 | com.alibaba.dubbo.rpc.cluster.loadbalance.LeastActiveLoadBalance |

**Dubbo服务集群容错实践**

手机应用是以聊天室为基础的，我们需要收集用户的操作行为，然后计算聊天室中在线人数，并实时在手机应用端显示人数，整个系统的架构如图所示：  
  
上图中，主要包括了两大主要流程：日志收集并实时处理流程、调用读取实时计算结果流程，我们使用基于Dubbo框架开发的服务来提供实时计算结果读取聊天人数的功能。上图中，实际上业务接口服务器集群也可以基于Dubbo框架构建服务，就看我们想要构建什么样的系统来满足我们的需要。  
如果不使用注册中心，服务消费方也能够直接调用服务提供方发布的服务，这样需要服务提供方将服务地址暴露给服务消费方，而且也无法使用监控中心的功能，这种方式成为直连。  
如果我们使用注册中心，服务提供方将服务发布到注册中心，而服务消费方可以通过注册中心订阅服务，接收服务提供方服务变更通知，这种方式可以隐藏服务提供方的细节，包括服务器地址等敏感信息，而服务消费方只能通过注册中心来获取到已注册的提供方服务，而不能直接跨过注册中心与服务提供方直接连接。这种方式的好处是还可以使用监控中心服务，能够对服务的调用情况进行监控分析，还能使用Dubbo服务管理中心，方便管理服务，我们在这里使用的是这种方式，也推荐使用这种方式。使用注册中心的Dubbo分布式服务相关组件结构，如下图所示：

下面，开发部署我们的应用，通过如下4个步骤来完成：

* 服务接口定义

服务接口将服务提供方（Provider）和服务消费方（Consumer）连接起来，服务提供方实现接口中定义的服务，即给出服务的实现，而服务消费方负责调用服务。我们接口中给出了2个方法，一个是实时查询获取当前聊天室内人数，另一个是查询一天中某个/某些聊天室中在线人数峰值，接口定义如下所示：

|  |  |
| --- | --- |
| 01 | **package** org.shirdrn.dubbo.api; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | **import** java.util.List; |

|  |  |
| --- | --- |
| 04 |  |

|  |  |
| --- | --- |
| 05 | **public** **interface** ChatRoomOnlineUserCounterService { |

|  |  |
| --- | --- |
| 06 |  |

|  |  |
| --- | --- |
| 07 | String queryRoomUserCount(String rooms); |

|  |  |
| --- | --- |
| 08 |  |

|  |  |
| --- | --- |
| 09 | List<String> getMaxOnlineUserCount(List<String> rooms, String date, String dateFormat); |

|  |  |
| --- | --- |
| 10 | } |

接口是服务提供方和服务消费方公共遵守的协议，一般情况下是服务提供方将接口定义好后提供给服务消费方。

* 服务提供方

服务提供方实现接口中定义的服务，其实现和普通的服务没什么区别，我们的实现类为ChatRoomOnlineUserCounterServiceImpl，代码如下所示：

|  |  |
| --- | --- |
| 01 | **package** org.shirdrn.dubbo.provider.service; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | **import** java.util.List; |

|  |  |
| --- | --- |
| 04 |  |

|  |  |
| --- | --- |
| 05 | **import** org.apache.commons.logging.Log; |

|  |  |
| --- | --- |
| 06 | **import** org.apache.commons.logging.LogFactory; |

|  |  |
| --- | --- |
| 07 | **import** org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService; |

|  |  |
| --- | --- |
| 08 | **import** org.shirdrn.dubbo.common.utils.DateTimeUtils; |

|  |  |
| --- | --- |
| 09 |  |

|  |  |
| --- | --- |
| 10 | **import** redis.clients.jedis.Jedis; |

|  |  |
| --- | --- |
| 11 | **import** redis.clients.jedis.JedisPool; |

|  |  |
| --- | --- |
| 12 |  |

|  |  |
| --- | --- |
| 13 | **import** com.alibaba.dubbo.common.utils.StringUtils; |

|  |  |
| --- | --- |
| 14 | **import** com.google.common.base.Strings; |

|  |  |
| --- | --- |
| 15 | **import** com.google.common.collect.Lists; |

|  |  |
| --- | --- |
| 16 |  |

|  |  |
| --- | --- |
| 17 | **public** **class** ChatRoomOnlineUserCounterServiceImpl **implements** ChatRoomOnlineUserCounterService { |

|  |  |
| --- | --- |
| 18 |  |

|  |  |
| --- | --- |
| 19 | **private** **static** **final** Log LOG = LogFactory.getLog(ChatRoomOnlineUserCounterServiceImpl.**class**); |

|  |  |
| --- | --- |
| 20 | **private** JedisPool jedisPool; |

|  |  |
| --- | --- |
| 21 | **private** **static** **final** String KEY\_USER\_COUNT = "chat::room::play::user::cnt"; |

|  |  |
| --- | --- |
| 22 | **private** **static** **final** String KEY\_MAX\_USER\_COUNT\_PREFIX = "chat::room::max::user::cnt::"; |

|  |  |
| --- | --- |
| 23 | **private** **static** **final** String DF\_YYYYMMDD = "yyyyMMdd"; |

|  |  |
| --- | --- |
| 24 |  |

|  |  |
| --- | --- |
| 25 | **public** String queryRoomUserCount(String rooms) { |

|  |  |
| --- | --- |
| 26 | LOG.info("Params[Server|Recv|REQ] rooms=" + rooms); |

|  |  |
| --- | --- |
| 27 | StringBuffer builder = **new** StringBuffer(); |

|  |  |
| --- | --- |
| 28 | **if**(!Strings.isNullOrEmpty(rooms)) { |

|  |  |
| --- | --- |
| 29 | Jedis jedis = **null**; |

|  |  |
| --- | --- |
| 30 | **try** { |

|  |  |
| --- | --- |
| 31 | jedis = jedisPool.getResource(); |

|  |  |
| --- | --- |
| 32 | String[] fields = rooms.split(","); |

|  |  |
| --- | --- |
| 33 | List<String> results = jedis.hmget(KEY\_USER\_COUNT, fields); |

|  |  |
| --- | --- |
| 34 | builder.append(StringUtils.join(results, ",")); |

|  |  |
| --- | --- |
| 35 | } **catch** (Exception e) { |

|  |  |
| --- | --- |
| 36 | LOG.error("", e); |

|  |  |
| --- | --- |
| 37 | } **finally** { |

|  |  |
| --- | --- |
| 38 | **if**(jedis != **null**) { |

|  |  |
| --- | --- |
| 39 | jedis.close(); |

|  |  |
| --- | --- |
| 40 | } |

|  |  |
| --- | --- |
| 41 | } |

|  |  |
| --- | --- |
| 42 | } |

|  |  |
| --- | --- |
| 43 | LOG.info("Result[Server|Recv|RES] " + builder.toString()); |

|  |  |
| --- | --- |
| 44 | **return** builder.toString(); |

|  |  |
| --- | --- |
| 45 | } |

|  |  |
| --- | --- |
| 46 |  |

|  |  |
| --- | --- |
| 47 | @Override |

|  |  |
| --- | --- |
| 48 | **public** List<String> getMaxOnlineUserCount(List<String> rooms, String date, String dateFormat) { |

|  |  |
| --- | --- |
| 49 | // HGETALL chat::room::max::user::cnt::20150326 |

|  |  |
| --- | --- |
| 50 | LOG.info("Params[Server|Recv|REQ] rooms=" + rooms + ",date=" + date + ",dateFormat=" + dateFormat); |

|  |  |
| --- | --- |
| 51 | String whichDate = DateTimeUtils.format(date, dateFormat, DF\_YYYYMMDD); |

|  |  |
| --- | --- |
| 52 | String key = KEY\_MAX\_USER\_COUNT\_PREFIX + whichDate; |

|  |  |
| --- | --- |
| 53 | StringBuffer builder = **new** StringBuffer(); |

|  |  |
| --- | --- |
| 54 | **if**(rooms != **null** && !rooms.isEmpty()) { |

|  |  |
| --- | --- |
| 55 | Jedis jedis = **null**; |

|  |  |
| --- | --- |
| 56 | **try** { |

|  |  |
| --- | --- |
| 57 | jedis = jedisPool.getResource(); |

|  |  |
| --- | --- |
| 58 | **return** jedis.hmget(key, rooms.toArray(**new** String[rooms.size()])); |

|  |  |
| --- | --- |
| 59 | } **catch** (Exception e) { |

|  |  |
| --- | --- |
| 60 | LOG.error("", e); |

|  |  |
| --- | --- |
| 61 | } **finally** { |

|  |  |
| --- | --- |
| 62 | **if**(jedis != **null**) { |

|  |  |
| --- | --- |
| 63 | jedis.close(); |

|  |  |
| --- | --- |
| 64 | } |

|  |  |
| --- | --- |
| 65 | } |

|  |  |
| --- | --- |
| 66 | } |

|  |  |
| --- | --- |
| 67 | LOG.info("Result[Server|Recv|RES] " + builder.toString()); |

|  |  |
| --- | --- |
| 68 | **return** Lists.newArrayList(); |

|  |  |
| --- | --- |
| 69 | } |

|  |  |
| --- | --- |
| 70 |  |

|  |  |
| --- | --- |
| 71 | **public** **void** setJedisPool(JedisPool jedisPool) { |

|  |  |
| --- | --- |
| 72 | **this**.jedisPool = jedisPool; |

|  |  |
| --- | --- |
| 73 | } |

|  |  |
| --- | --- |
| 74 |  |

|  |  |
| --- | --- |
| 75 | } |

代码中通过读取Redis中数据来完成调用，逻辑比较简单。对应的Maven POM依赖配置，如下所示：

|  |  |
| --- | --- |
| 01 | <**dependencies**> |

|  |  |
| --- | --- |
| 02 | <**dependency**> |

|  |  |
| --- | --- |
| 03 | <**groupId**>org.shirdrn.dubbo</**groupId**> |

|  |  |
| --- | --- |
| 04 | <**artifactId**>dubbo-api</**artifactId**> |

|  |  |
| --- | --- |
| 05 | <**version**>0.0.1-SNAPSHOT</**version**> |

|  |  |
| --- | --- |
| 06 | </**dependency**> |

|  |  |
| --- | --- |
| 07 | <**dependency**> |

|  |  |
| --- | --- |
| 08 | <**groupId**>org.shirdrn.dubbo</**groupId**> |

|  |  |
| --- | --- |
| 09 | <**artifactId**>dubbo-commons</**artifactId**> |

|  |  |
| --- | --- |
| 10 | <**version**>0.0.1-SNAPSHOT</**version**> |

|  |  |
| --- | --- |
| 11 | </**dependency**> |

|  |  |
| --- | --- |
| 12 | <**dependency**> |

|  |  |
| --- | --- |
| 13 | <**groupId**>redis.clients</**groupId**> |

|  |  |
| --- | --- |
| 14 | <**artifactId**>jedis</**artifactId**> |

|  |  |
| --- | --- |
| 15 | <**version**>2.5.2</**version**> |

|  |  |
| --- | --- |
| 16 | </**dependency**> |

|  |  |
| --- | --- |
| 17 | <**dependency**> |

|  |  |
| --- | --- |
| 18 | <**groupId**>org.apache.commons</**groupId**> |

|  |  |
| --- | --- |
| 19 | <**artifactId**>commons-pool2</**artifactId**> |

|  |  |
| --- | --- |
| 20 | <**version**>2.2</**version**> |

|  |  |
| --- | --- |
| 21 | </**dependency**> |

|  |  |
| --- | --- |
| 22 | <**dependency**> |

|  |  |
| --- | --- |
| 23 | <**groupId**>org.jboss.netty</**groupId**> |

|  |  |
| --- | --- |
| 24 | <**artifactId**>netty</**artifactId**> |

|  |  |
| --- | --- |
| 25 | <**version**>3.2.7.Final</**version**> |

|  |  |
| --- | --- |
| 26 | </**dependency**> |

|  |  |
| --- | --- |
| 27 | </**dependencies**> |

有关对Dubbo框架的一些依赖，我们单独放到一个通用的Maven Module中（详见后面“附录：Dubbo使用Maven构建依赖配置”），这里不再多说。服务提供方实现，最关键的就是服务的配置，因为Dubbo基于Spring来管理配置和实例，所以通过配置可以指定服务是否是分布式服务，以及通过配置增加很多其它特性。我们的配置文件为provider-cluster.xml，内容如下所示：

|  |  |
| --- | --- |
| 01 | <?**xml** version="1.0" encoding="UTF-8"?> |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | <**beans** xmlns="<http://www.springframework.org/schema/beans>" |

|  |  |
| --- | --- |
| 04 | xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns:dubbo="<http://code.alibabatech.com/schema/dubbo>" |

|  |  |
| --- | --- |
| 05 | xmlns:p="<http://www.springframework.org/schema/p>" |

|  |  |
| --- | --- |
| 06 | xsi:schemaLocation="<http://www.springframework.org/schema/beans><http://www.springframework.org/schema/beans/spring-beans-3.0.xsd> |

|  |  |
| --- | --- |
| 07 | <http://code.alibabatech.com/schema/dubbo> <http://code.alibabatech.com/schema/dubbo/dubbo.xsd>"> |

|  |  |
| --- | --- |
| 08 |  |

|  |  |
| --- | --- |
| 09 | <**bean** class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"> |

|  |  |
| --- | --- |
| 10 | <**property** name="systemPropertiesModeName" value="SYSTEM\_PROPERTIES\_MODE\_OVERRIDE" /> |

|  |  |
| --- | --- |
| 11 | <**property** name="ignoreResourceNotFound" value="true" /> |

|  |  |
| --- | --- |
| 12 | <**property** name="locations"> |

|  |  |
| --- | --- |
| 13 | <**list**> |

|  |  |
| --- | --- |
| 14 | <**value**>classpath\*:jedis.properties</**value**> |

|  |  |
| --- | --- |
| 15 | </**list**> |

|  |  |
| --- | --- |
| 16 | </**property**> |

|  |  |
| --- | --- |
| 17 | </**bean**> |

|  |  |
| --- | --- |
| 18 |  |

|  |  |
| --- | --- |
| 19 | <**dubbo:application** name="chatroom-cluster-provider" /> |

|  |  |
| --- | --- |
| 20 | <**dubbo:registry** address="<zookeeper://zk1:2181?backup=zk2:2181>,zk3:2181" /> |

|  |  |
| --- | --- |
| 21 |  |

|  |  |
| --- | --- |
| 22 | <**dubbo:protocol** name="dubbo" port="20880" /> |

|  |  |
| --- | --- |
| 23 |  |

|  |  |
| --- | --- |
| 24 | <**dubbo:service** interface="org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService" version="1.0.0" |

|  |  |
| --- | --- |
| 25 | cluster="failover" retries="2" timeout="1000" loadbalance="random" actives="100" executes="200" |

|  |  |
| --- | --- |
| 26 | ref="chatRoomOnlineUserCounterService" protocol="dubbo" > |

|  |  |
| --- | --- |
| 27 | <**dubbo:method** name="queryRoomUserCount" timeout="500" retries="2" loadbalance="roundrobin" actives="50"/> |

|  |  |
| --- | --- |
| 28 | </**dubbo:service**> |

|  |  |
| --- | --- |
| 29 |  |

|  |  |
| --- | --- |
| 30 | <**bean** id="chatRoomOnlineUserCounterService"class="org.shirdrn.dubbo.provider.service.ChatRoomOnlineUserCounterServiceImpl" > |

|  |  |
| --- | --- |
| 31 | <**property** name="jedisPool" ref="jedisPool" /> |

|  |  |
| --- | --- |
| 32 | </**bean**> |

|  |  |
| --- | --- |
| 33 |  |

|  |  |
| --- | --- |
| 34 | <**bean** id="jedisPool" class="redis.clients.jedis.JedisPool" destroy-method="destroy"> |

|  |  |
| --- | --- |
| 35 | <**constructor-arg** index="0"> |

|  |  |
| --- | --- |
| 36 | <**bean** class="org.apache.commons.pool2.impl.GenericObjectPoolConfig"> |

|  |  |
| --- | --- |
| 37 | <**property** name="maxTotal" value="${redis.pool.maxTotal}" /> |

|  |  |
| --- | --- |
| 38 | <**property** name="maxIdle" value="${redis.pool.maxIdle}" /> |

|  |  |
| --- | --- |
| 39 | <**property** name="minIdle" value="${redis.pool.minIdle}" /> |

|  |  |
| --- | --- |
| 40 | <**property** name="maxWaitMillis" value="${redis.pool.maxWaitMillis}" /> |

|  |  |
| --- | --- |
| 41 | <**property** name="testOnBorrow" value="${redis.pool.testOnBorrow}" /> |

|  |  |
| --- | --- |
| 42 | <**property** name="testOnReturn" value="${redis.pool.testOnReturn}" /> |

|  |  |
| --- | --- |
| 43 | <**property** name="testWhileIdle" value="true" /> |

|  |  |
| --- | --- |
| 44 | </**bean**> |

|  |  |
| --- | --- |
| 45 | </**constructor-arg**> |

|  |  |
| --- | --- |
| 46 | <**constructor-arg** index="1" value="${redis.host}" /> |

|  |  |
| --- | --- |
| 47 | <**constructor-arg** index="2" value="${redis.port}" /> |

|  |  |
| --- | --- |
| 48 | <**constructor-arg** index="3" value="${redis.timeout}" /> |

|  |  |
| --- | --- |
| 49 | </**bean**> |

|  |  |
| --- | --- |
| 50 |  |

|  |  |
| --- | --- |
| 51 | </**beans**> |

上面配置中，使用dubbo协议，集群容错模式为failover，服务级别负载均衡策略为random，方法级别负载均衡策略为roundrobin（它覆盖了服务级别的配置内容），其他一些配置内容可以参考Dubbo文档。我们这里是从Redis读取数据，所以使用了Redis连接池。  
启动服务示例代码如下所示：

|  |  |
| --- | --- |
| 01 | **package** org.shirdrn.dubbo.provider; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | **import** org.shirdrn.dubbo.provider.common.DubboServer; |

|  |  |
| --- | --- |
| 04 |  |

|  |  |
| --- | --- |
| 05 | **public** **class** ChatRoomClusterServer { |

|  |  |
| --- | --- |
| 06 |  |

|  |  |
| --- | --- |
| 07 | **public** **static** **void** main(String[] args) **throws** Exception { |

|  |  |
| --- | --- |
| 08 | DubboServer.startServer("classpath:provider-cluster.xml"); |

|  |  |
| --- | --- |
| 09 | } |

|  |  |
| --- | --- |
| 10 |  |

|  |  |
| --- | --- |
| 11 | } |

上面调用了DubboServer类的静态方法startServer，如下所示：

|  |  |
| --- | --- |
| 01 | **public** **static** **void** startServer(String config) { |

|  |  |
| --- | --- |
| 02 | ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(config); |

|  |  |
| --- | --- |
| 03 | **try** { |

|  |  |
| --- | --- |
| 04 | context.start(); |

|  |  |
| --- | --- |
| 05 | System.in.read(); |

|  |  |
| --- | --- |
| 06 | } **catch** (IOException e) { |

|  |  |
| --- | --- |
| 07 | e.printStackTrace(); |

|  |  |
| --- | --- |
| 08 | } **finally** { |

|  |  |
| --- | --- |
| 09 | context.close(); |

|  |  |
| --- | --- |
| 10 | } |

|  |  |
| --- | --- |
| 11 | } |

方法中主要是初始化Spring IoC容器，全部对象都交由容器来管理。

* 服务消费方

服务消费方就容易了，只需要知道注册中心地址，并引用服务提供方提供的接口，消费方调用服务实现如下所示：

|  |  |
| --- | --- |
| 01 | **package** org.shirdrn.dubbo.consumer; |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | **import** java.util.Arrays; |

|  |  |
| --- | --- |
| 04 | **import** java.util.List; |

|  |  |
| --- | --- |
| 05 |  |

|  |  |
| --- | --- |
| 06 | **import** org.apache.commons.logging.Log; |

|  |  |
| --- | --- |
| 07 | **import** org.apache.commons.logging.LogFactory; |

|  |  |
| --- | --- |
| 08 | **import** org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService; |

|  |  |
| --- | --- |
| 09 | **import** org.springframework.context.support.AbstractXmlApplicationContext; |

|  |  |
| --- | --- |
| 10 | **import** org.springframework.context.support.ClassPathXmlApplicationContext; |

|  |  |
| --- | --- |
| 11 |  |

|  |  |
| --- | --- |
| 12 | **public** **class** ChatRoomDubboConsumer { |

|  |  |
| --- | --- |
| 13 |  |

|  |  |
| --- | --- |
| 14 | **private** **static** **final** Log LOG = LogFactory.getLog(ChatRoomDubboConsumer.**class**); |

|  |  |
| --- | --- |
| 15 |  |

|  |  |
| --- | --- |
| 16 | **public** **static** **void** main(String[] args) **throws** Exception { |

|  |  |
| --- | --- |
| 17 | AbstractXmlApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:consumer.xml"); |

|  |  |
| --- | --- |
| 18 | **try** { |

|  |  |
| --- | --- |
| 19 | context.start(); |

|  |  |
| --- | --- |
| 20 | ChatRoomOnlineUserCounterService chatRoomOnlineUserCounterService = (ChatRoomOnlineUserCounterService) context.getBean("chatRoomOnlineUserCounterService"); |

|  |  |
| --- | --- |
| 21 | getMaxOnlineUserCount(chatRoomOnlineUserCounterService); |

|  |  |
| --- | --- |
| 22 | getRealtimeOnlineUserCount(chatRoomOnlineUserCounterService); |

|  |  |
| --- | --- |
| 23 | System.in.read(); |

|  |  |
| --- | --- |
| 24 | } **finally** { |

|  |  |
| --- | --- |
| 25 | context.close(); |

|  |  |
| --- | --- |
| 26 | } |

|  |  |
| --- | --- |
| 27 |  |

|  |  |
| --- | --- |
| 28 | } |

|  |  |
| --- | --- |
| 29 |  |

|  |  |
| --- | --- |
| 30 | **private** **static** **void** getMaxOnlineUserCount(ChatRoomOnlineUserCounterService liveRoomOnlineUserCountService) { |

|  |  |
| --- | --- |
| 31 | List<String> maxUserCounts = liveRoomOnlineUserCountService.getMaxOnlineUserCount( |

|  |  |
| --- | --- |
| 32 | Arrays.asList(**new** String[] {"1482178010" , "1408492761", "1430546839", "1412517075","1435861734"}), "20150327", "yyyyMMdd"); |

|  |  |
| --- | --- |
| 33 | LOG.info("After getMaxOnlineUserCount invoked: maxUserCounts= " + maxUserCounts); |

|  |  |
| --- | --- |
| 34 | } |

|  |  |
| --- | --- |
| 35 |  |

|  |  |
| --- | --- |
| 36 | **private** **static** **void** getRealtimeOnlineUserCount(ChatRoomOnlineUserCounterService liveRoomOnlineUserCountService) |

|  |  |
| --- | --- |
| 37 | **throws** InterruptedException { |

|  |  |
| --- | --- |
| 38 | String rooms = "1482178010,1408492761,1430546839,1412517075,1435861734"; |

|  |  |
| --- | --- |
| 39 | String onlineUserCounts = liveRoomOnlineUserCountService.queryRoomUserCount(rooms); |

|  |  |
| --- | --- |
| 40 | LOG.info("After queryRoomUserCount invoked: onlineUserCounts= " + onlineUserCounts); |

|  |  |
| --- | --- |
| 41 | } |

|  |  |
| --- | --- |
| 42 | } |

对应的配置文件为consumer.xml，内容如下所示：

|  |  |
| --- | --- |
| 01 | <?**xml** version="1.0" encoding="UTF-8"?> |

|  |  |
| --- | --- |
| 02 |  |

|  |  |
| --- | --- |
| 03 | <**beans** xmlns="<http://www.springframework.org/schema/beans>" |

|  |  |
| --- | --- |
| 04 | xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns:dubbo="<http://code.alibabatech.com/schema/dubbo>" |

|  |  |
| --- | --- |
| 05 | xsi:schemaLocation="<http://www.springframework.org/schema/beans><http://www.springframework.org/schema/beans/spring-beans-3.0.xsd> |

|  |  |
| --- | --- |
| 06 | <http://code.alibabatech.com/schema/dubbo> <http://code.alibabatech.com/schema/dubbo/dubbo.xsd>"> |

|  |  |
| --- | --- |
| 07 |  |

|  |  |
| --- | --- |
| 08 | <**dubbo:application** name="chatroom-consumer" /> |

|  |  |
| --- | --- |
| 09 | <**dubbo:registry** address="<zookeeper://zk1:2181?backup=zk2:2181>,zk3:2181" /> |

|  |  |
| --- | --- |
| 10 |  |

|  |  |
| --- | --- |
| 11 | <**dubbo:reference** id="chatRoomOnlineUserCounterService"interface="org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService" version="1.0.0"> |

|  |  |
| --- | --- |
| 12 | <**dubbo:method** name="queryRoomUserCount" retries="2" /> |

|  |  |
| --- | --- |
| 13 | </**dubbo:reference**> |

|  |  |
| --- | --- |
| 14 |  |

|  |  |
| --- | --- |
| 15 | </**beans**> |

也可以根据需要配置dubbo:reference相关的属性值，也可以配置dubbo:method指定调用的方法的配置信息，详细配置属性可以参考Dubbo官方文档。

* 部署与验证

开发完成提供方服务后，在本地开发调试的时候可以怎么简单怎么做，如果是要部署到生产环境，则需要打包后进行部署，可以参考下面的Maven POM配置：

|  |  |
| --- | --- |
| 01 | <**build**> |

|  |  |
| --- | --- |
| 02 | <**plugins**> |

|  |  |
| --- | --- |
| 03 | <**plugin**> |

|  |  |
| --- | --- |
| 04 | <**groupId**>org.apache.maven.plugins</**groupId**> |

|  |  |
| --- | --- |
| 05 | <**artifactId**>maven-shade-plugin</**artifactId**> |

|  |  |
| --- | --- |
| 06 | <**version**>1.4</**version**> |

|  |  |
| --- | --- |
| 07 | <**configuration**> |

|  |  |
| --- | --- |
| 08 | <**createDependencyReducedPom**>true</**createDependencyReducedPom**> |

|  |  |
| --- | --- |
| 09 | </**configuration**> |

|  |  |
| --- | --- |
| 10 | <**executions**> |

|  |  |
| --- | --- |
| 11 | <**execution**> |

|  |  |
| --- | --- |
| 12 | <**phase**>package</**phase**> |

|  |  |
| --- | --- |
| 13 | <**goals**> |

|  |  |
| --- | --- |
| 14 | <**goal**>shade</**goal**> |

|  |  |
| --- | --- |
| 15 | </**goals**> |

|  |  |
| --- | --- |
| 16 | <**configuration**> |

|  |  |
| --- | --- |
| 17 | <**transformers**> |

|  |  |
| --- | --- |
| 18 | <**transformer**implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer" /> |

|  |  |
| --- | --- |
| 19 | <**transformer**implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"> |

|  |  |
| --- | --- |
| 20 | <**mainClass**>org.shirdrn.dubbo.provider.ChatRoomClusterServer</**mainClass**> |

|  |  |
| --- | --- |
| 21 | </**transformer**> |

|  |  |
| --- | --- |
| 22 | </**transformers**> |

|  |  |
| --- | --- |
| 23 | </**configuration**> |

|  |  |
| --- | --- |
| 24 | </**execution**> |

|  |  |
| --- | --- |
| 25 | </**executions**> |

|  |  |
| --- | --- |
| 26 | </**plugin**> |

|  |  |
| --- | --- |
| 27 | </**plugins**> |

|  |  |
| --- | --- |
| 28 | </**build**> |

这里也给出Maven POM依赖的简单配置：

|  |  |
| --- | --- |
| 1 | <**dependencies**> |

|  |  |
| --- | --- |
| 2 | <**dependency**> |

|  |  |
| --- | --- |
| 3 | <**groupId**>org.shirdrn.dubbo</**groupId**> |

|  |  |
| --- | --- |
| 4 | <**artifactId**>dubbo-api</**artifactId**> |

|  |  |
| --- | --- |
| 5 | <**version**>0.0.1-SNAPSHOT</**version**> |

|  |  |
| --- | --- |
| 6 | </**dependency**> |

|  |  |
| --- | --- |
| 7 | </**dependencies**> |

我们开发的服务应该是分布式的，首先是通过配置内容来决定，例如设置集群模式、设置负载均衡模式等，然后在部署的时候，可以在多个节点上同一个服务，这样多个服务都会注册到Dubbo注册中心，如果某个节点上的服务不可用了，可以根据我们配置的策略来选择其他节点上的可用服务，后面通过Dubbo服务管理中心和监控中心就能更加清楚明了。

**Dubbo服务管理与监控**

我们需要在安装好管理中心和监控中心以后，再将上面的开发的提供方服务部署到物理节点上，然后就能够通过管理中心和监控中心来查看对应的详细情况。

* Dubbo服务管理中心

安装Dubbo服务管理中心，需要选择一个Web容器，我们使用Tomcat服务器。首先下载Dubbo管理中心安装文件dubbo-admin-2.5.3.war，或者直接从源码构建得到该WAR文件。这里，我们已经构建好对应的WAR文件，然后进行安装，执行如下命令：

|  |  |
| --- | --- |
| 1 | cd apache-tomcat-6.0.35 |

|  |  |
| --- | --- |
| 2 | rm -rf webapps/ROOT |

|  |  |
| --- | --- |
| 3 | unzip ~/dubbo-admin-2.5.3.war -d webapps/ROOT |

修改配置文件~/apache-tomcat-6.0.35/webapps/ROOT/WEB-INF/dubbo.properties，指定我们的注册中心地址以及登录密码，内容如下所示：

|  |  |
| --- | --- |
| 1 | dubbo.registry.address=<zookeeper://zk1:2181?backup=zk2:2181>,zk3:2181 |

|  |  |
| --- | --- |
| 2 | dubbo.admin.root.password=root |

|  |  |
| --- | --- |
| 3 | dubbo.admin.guest.password=guest |

然后，根据需要修改~/apache-tomcat-6.0.35/conf/server.xml配置文件，主要是Tomcat HTTP 端口号（我这里使用8083端口），完成后可以直接启动Tomcat服务器：

|  |  |
| --- | --- |
| 1 | cd ~/apache-tomcat-6.0.35/ |

|  |  |
| --- | --- |
| 2 | bin/catalina.sh start |

然后访问地址<http://10.10.4.130:8083/>即可，根据配置文件指定的root用户密码，就可以登录Dubbo管理控制台。  
我们将上面开发的服务提供方服务，部署到2个独立的节点上（192.168.14.1和10.10.4.125），然后可以通过Dubbo管理中心查看对应服务的状况，如图所示：  
  
上图中可以看出，该服务有两个独立的节点可以提供，因为配置的集群模式为failover，如果某个节点的服务发生故障无法使用，则会自动透明地重试另一个节点上的服务，这样就不至于出现拒绝服务的情况。如果想要查看提供方某个节点上的服务详情，可以点击对应的IP:Port链接，示例如图所示：  
  
上图可以看到服务地址：

|  |  |
| --- | --- |
| 1 | <dubbo://10.10.4.125:20880/org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService?actives=100&anyhost=true&application=chatroom-cluster-provider&cluster=failover&dubbo=0.0.1-SNAPSHOT&executes=200&interface=org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService&loadbalance=random&methods=getMaxOnlineUserCount>,queryRoomUserCount&pid=30942&queryRoomUserCount.actives=50&queryRoomUserCount.loadbalance=leastactive&queryRoomUserCount.retries=2&queryRoomUserCount.timeout=500&retries=2&revision=0.0.1-SNAPSHOT&side=provider&timeout=1000×tamp=1427793652814&version=1.0.0 |

如果我们直接暴露该地址也是可以的，不过这种直连的方式对服务消费方不是透明的，如果以后IP地址更换，也会影响调用方，所以最好是通过注册中心来隐蔽服务地址。同一个服务所部署在的多个节点上，也就对应对应着多个服务地址。另外，也可以对已经发布的服务进行控制，如修改访问控制、负载均衡相关配置内容等，可以通过上图中“消费者”查看服务消费方调用服务的情况，如图所示：  
  
也在管理控制台可以对消费方进行管理控制。

* Dubbo监控中心

Dubbo监控中心是以Dubbo服务的形式发布到注册中心，和普通的服务时一样的。例如，我这里下载了Dubbo自带的简易监控中心文件dubbo-monitor-simple-2.5.3-assembly.tar.gz，可以解压缩以后，修改配置文件~/dubbo-monitor-simple-2.5.3/conf/dubbo.properties的内容，如下所示：

|  |  |
| --- | --- |
| 01 | dubbo.container=log4j,spring,registry,jetty |

|  |  |
| --- | --- |
| 02 | dubbo.application.name=simple-monitor |

|  |  |
| --- | --- |
| 03 | dubbo.application.owner= |

|  |  |
| --- | --- |
| 04 | dubbo.registry.address=<zookeeper://zk1:2181?backup=zk2:2181>,zk3:2181 |

|  |  |
| --- | --- |
| 05 | dubbo.protocol.port=7070 |

|  |  |
| --- | --- |
| 06 | dubbo.jetty.port=8087 |

|  |  |
| --- | --- |
| 07 | dubbo.jetty.directory=${user.home}/monitor |

|  |  |
| --- | --- |
| 08 | dubbo.charts.directory=${dubbo.jetty.directory}/charts |

|  |  |
| --- | --- |
| 09 | dubbo.statistics.directory=${user.home}/monitor/statistics |

|  |  |
| --- | --- |
| 10 | dubbo.log4j.file=logs/dubbo-monitor-simple.log |

|  |  |
| --- | --- |
| 11 | dubbo.log4j.level=WARN |

然后启动简易监控中心，执行如下命令：

|  |  |
| --- | --- |
| 1 | cd ~/dubbo-monitor-simple-2.5.3 |

|  |  |
| --- | --- |
| 2 | bin/start.sh |

这里使用了Jetty Web容器，访问地址<http://10.10.4.130:8087/>就可以查看监控中心，Applications选项卡页面包含了服务提供方和消费方的基本信息，如图所示：  
  
上图主要列出了所有提供方发布的服务、消费方调用、服务依赖关系等内容。  
接着，查看Services选项卡页面，包含了服务提供方提供的服务列表，如图所示：  
  
点击上图中Providers链接就能看到服务提供方的基本信息，包括服务地址等，如图所示：  
  
点击上图中Consumers链接就能看到服务消费方的基本信息，包括服务地址等，如图所示：  
  
由于上面是Dubbo自带的一个简易监控中心，可能所展现的内容并不能满足我们的需要，所以可以根据需要开发自己的监控中心。Dubbo也提供了监控中心的扩展接口，如果想要实现自己的监控中心，可以实现接口com.alibaba.dubbo.monitor.MonitorFactory和com.alibaba.dubbo.monitor.Monitor，其中MonitorFactory接口定义如下所示：

|  |  |
| --- | --- |
| 01 | /\*\* |

|  |  |
| --- | --- |
| 02 | \* MonitorFactory. (SPI, Singleton, ThreadSafe) |

|  |  |
| --- | --- |
| 03 | \* |

|  |  |
| --- | --- |
| 04 | \* @author william.liangf |

|  |  |
| --- | --- |
| 05 | \*/ |

|  |  |
| --- | --- |
| 06 | @SPI("dubbo") |

|  |  |
| --- | --- |
| 07 | **public** **interface** MonitorFactory { |

|  |  |
| --- | --- |
| 08 |  |

|  |  |
| --- | --- |
| 09 | /\*\* |

|  |  |
| --- | --- |
| 10 | \* Create monitor. |

|  |  |
| --- | --- |
| 11 | \* @param url |

|  |  |
| --- | --- |
| 12 | \* @return monitor |

|  |  |
| --- | --- |
| 13 | \*/ |

|  |  |
| --- | --- |
| 14 | @Adaptive("protocol") |

|  |  |
| --- | --- |
| 15 | Monitor getMonitor(URL url); |

|  |  |
| --- | --- |
| 16 |  |

|  |  |
| --- | --- |
| 17 | } |

Monitor接口定义如下所示：

|  |  |
| --- | --- |
| 1 | /\*\* |

|  |  |
| --- | --- |
| 2 | \* Monitor. (SPI, Prototype, ThreadSafe) |

|  |  |
| --- | --- |
| 3 | \* |

|  |  |
| --- | --- |
| 4 | \* @see com.alibaba.dubbo.monitor.MonitorFactory#getMonitor(com.alibaba.dubbo.common.URL) |

|  |  |
| --- | --- |
| 5 | \* @author william.liangf |

|  |  |
| --- | --- |
| 6 | \*/ |

|  |  |
| --- | --- |
| 7 | **public** **interface** Monitor **extends** Node, MonitorService { |

|  |  |
| --- | --- |
| 8 |  |

|  |  |
| --- | --- |
| 9 | } |

具体定义内容可以查看MonitorService接口，不再累述。

**总结**

Dubbo还提供了其他很多高级特性，如路由规则、参数回调、服务分组、服务降级等等，而且很多特性在给出内置实现的基础上，还给出了扩展的接口，我们可以给出自定义的实现，非常方便而且强大。更多可以参考Dubbo官网用户手册和开发手册。

**附录：Dubbo使用Maven构建依赖配置**

[查看源代码](http://shiyanjun.cn/archives/1075.html#viewSource)[打印](http://shiyanjun.cn/archives/1075.html#printSource)[帮助](http://shiyanjun.cn/archives/1075.html#about)

|  |  |
| --- | --- |
| 01 | <**properties**> |

|  |  |
| --- | --- |
| 02 | <**spring.version**>3.2.8.RELEASE</**spring.version**> |

|  |  |
| --- | --- |
| 03 | <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**> |

|  |  |
| --- | --- |
| 04 | </**properties**> |

|  |  |
| --- | --- |
| 05 |  |

|  |  |
| --- | --- |
| 06 | <**dependencies**> |

|  |  |
| --- | --- |
| 07 | <**dependency**> |

|  |  |
| --- | --- |
| 08 | <**groupId**>com.alibaba</**groupId**> |

|  |  |
| --- | --- |
| 09 | <**artifactId**>dubbo</**artifactId**> |

|  |  |
| --- | --- |
| 10 | <**version**>2.5.3</**version**> |

|  |  |
| --- | --- |
| 11 | <**exclusions**> |

|  |  |
| --- | --- |
| 12 | <**exclusion**> |

|  |  |
| --- | --- |
| 13 | <**groupId**>org.springframework</**groupId**> |

|  |  |
| --- | --- |
| 14 | <**artifactId**>spring</**artifactId**> |

|  |  |
| --- | --- |
| 15 | </**exclusion**> |

|  |  |
| --- | --- |
| 16 | <**exclusion**> |

|  |  |
| --- | --- |
| 17 | <**groupId**>org.apache.zookeeper</**groupId**> |

|  |  |
| --- | --- |
| 18 | <**artifactId**>zookeeper</**artifactId**> |

|  |  |
| --- | --- |
| 19 | </**exclusion**> |

|  |  |
| --- | --- |
| 20 | <**exclusion**> |

|  |  |
| --- | --- |
| 21 | <**groupId**>org.jboss.netty</**groupId**> |

|  |  |
| --- | --- |
| 22 | <**artifactId**>netty</**artifactId**> |

|  |  |
| --- | --- |
| 23 | </**exclusion**> |

|  |  |
| --- | --- |
| 24 | </**exclusions**> |

|  |  |
| --- | --- |
| 25 | </**dependency**> |

|  |  |
| --- | --- |
| 26 | <**dependency**> |

|  |  |
| --- | --- |
| 27 | <**groupId**>org.springframework</**groupId**> |

|  |  |
| --- | --- |
| 28 | <**artifactId**>spring-core</**artifactId**> |

|  |  |
| --- | --- |
| 29 | <**version**>${spring.version}</**version**> |

|  |  |
| --- | --- |
| 30 | </**dependency**> |

|  |  |
| --- | --- |
| 31 | <**dependency**> |

|  |  |
| --- | --- |
| 32 | <**groupId**>org.springframework</**groupId**> |

|  |  |
| --- | --- |
| 33 | <**artifactId**>spring-beans</**artifactId**> |

|  |  |
| --- | --- |
| 34 | <**version**>${spring.version}</**version**> |

|  |  |
| --- | --- |
| 35 | </**dependency**> |

|  |  |
| --- | --- |
| 36 | <**dependency**> |

|  |  |
| --- | --- |
| 37 | <**groupId**>org.springframework</**groupId**> |

|  |  |
| --- | --- |
| 38 | <**artifactId**>spring-context</**artifactId**> |

|  |  |
| --- | --- |
| 39 | <**version**>${spring.version}</**version**> |

|  |  |
| --- | --- |
| 40 | </**dependency**> |

|  |  |
| --- | --- |
| 41 | <**dependency**> |

|  |  |
| --- | --- |
| 42 | <**groupId**>org.springframework</**groupId**> |

|  |  |
| --- | --- |
| 43 | <**artifactId**>spring-context-support</**artifactId**> |

|  |  |
| --- | --- |
| 44 | <**version**>${spring.version}</**version**> |

|  |  |
| --- | --- |
| 45 | </**dependency**> |

|  |  |
| --- | --- |
| 46 | <**dependency**> |

|  |  |
| --- | --- |
| 47 | <**groupId**>org.springframework</**groupId**> |

|  |  |
| --- | --- |
| 48 | <**artifactId**>spring-web</**artifactId**> |

|  |  |
| --- | --- |
| 49 | <**version**>${spring.version}</**version**> |

|  |  |
| --- | --- |
| 50 | </**dependency**> |

|  |  |
| --- | --- |
| 51 |  |

|  |  |
| --- | --- |
| 52 | <**dependency**> |

|  |  |
| --- | --- |
| 53 | <**groupId**>org.slf4j</**groupId**> |

|  |  |
| --- | --- |
| 54 | <**artifactId**>slf4j-api</**artifactId**> |

|  |  |
| --- | --- |
| 55 | <**version**>1.6.2</**version**> |

|  |  |
| --- | --- |
| 56 | </**dependency**> |

|  |  |
| --- | --- |
| 57 | <**dependency**> |

|  |  |
| --- | --- |
| 58 | <**groupId**>log4j</**groupId**> |

|  |  |
| --- | --- |
| 59 | <**artifactId**>log4j</**artifactId**> |

|  |  |
| --- | --- |
| 60 | <**version**>1.2.16</**version**> |

|  |  |
| --- | --- |
| 61 | </**dependency**> |

|  |  |
| --- | --- |
| 62 | <**dependency**> |

|  |  |
| --- | --- |
| 63 | <**groupId**>org.javassist</**groupId**> |

|  |  |
| --- | --- |
| 64 | <**artifactId**>javassist</**artifactId**> |

|  |  |
| --- | --- |
| 65 | <**version**>3.15.0-GA</**version**> |

|  |  |
| --- | --- |
| 66 | </**dependency**> |

|  |  |
| --- | --- |
| 67 | <**dependency**> |

|  |  |
| --- | --- |
| 68 | <**groupId**>com.alibaba</**groupId**> |

|  |  |
| --- | --- |
| 69 | <**artifactId**>hessian-lite</**artifactId**> |

|  |  |
| --- | --- |
| 70 | <**version**>3.2.1-fixed-2</**version**> |

|  |  |
| --- | --- |
| 71 | </**dependency**> |

|  |  |
| --- | --- |
| 72 | <**dependency**> |

|  |  |
| --- | --- |
| 73 | <**groupId**>com.alibaba</**groupId**> |

|  |  |
| --- | --- |
| 74 | <**artifactId**>fastjson</**artifactId**> |

|  |  |
| --- | --- |
| 75 | <**version**>1.1.8</**version**> |

|  |  |
| --- | --- |
| 76 | </**dependency**> |

|  |  |
| --- | --- |
| 77 | <**dependency**> |

|  |  |
| --- | --- |
| 78 | <**groupId**>org.jvnet.sorcerer</**groupId**> |

|  |  |
| --- | --- |
| 79 | <**artifactId**>sorcerer-javac</**artifactId**> |

|  |  |
| --- | --- |
| 80 | <**version**>0.8</**version**> |

|  |  |
| --- | --- |
| 81 | </**dependency**> |

|  |  |
| --- | --- |
| 82 | <**dependency**> |

|  |  |
| --- | --- |
| 83 | <**groupId**>org.apache.zookeeper</**groupId**> |

|  |  |
| --- | --- |
| 84 | <**artifactId**>zookeeper</**artifactId**> |

|  |  |
| --- | --- |
| 85 | <**version**>3.4.5</**version**> |

|  |  |
| --- | --- |
| 86 | </**dependency**> |

|  |  |
| --- | --- |
| 87 | <**dependency**> |

|  |  |
| --- | --- |
| 88 | <**groupId**>com.github.sgroschupf</**groupId**> |

|  |  |
| --- | --- |
| 89 | <**artifactId**>zkclient</**artifactId**> |

|  |  |
| --- | --- |
| 90 | <**version**>0.1</**version**> |

|  |  |
| --- | --- |
| 91 | </**dependency**> |

|  |  |
| --- | --- |
| 92 | <**dependency**> |

|  |  |
| --- | --- |
| 93 | <**groupId**>org.jboss.netty</**groupId**> |

|  |  |
| --- | --- |
| 94 | <**artifactId**>netty</**artifactId**> |

|  |  |
| --- | --- |
| 95 | <**version**>3.2.7.Final</**version**> |

|  |  |
| --- | --- |
| 96 | </**dependency**> |

|  |  |
| --- | --- |
| 97 | </**dependencies**> |

# [Java 并发工具包 java.util.concurrent 用户指南](http://blog.csdn.net/defonds/article/details/44021605)

标签： [多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)[异步](http://www.csdn.net/tag/%e5%bc%82%e6%ad%a5)[java.util.concurrent](http://www.csdn.net/tag/java.util.concurrent)[Java多线程](http://www.csdn.net/tag/Java%e5%a4%9a%e7%ba%bf%e7%a8%8b)

2015-03-03 09:40 106657人阅读 [评论](http://blog.csdn.net/defonds/article/details/44021605/#comments)(18) [收藏](http://blog.csdn.net/defonds/article/details/44021605/javascript:void(0);) [举报](http://blog.csdn.net/defonds/article/details/44021605/#report)

 分类：

多线程编程（14）

[作者同类文章](http://blog.csdn.net/defonds/article/category/1363593)X

译序

本指南根据 Jakob Jenkov 最新博客翻译，请随时关注博客更新：<http://tutorials.jenkov.com/java-util-concurrent/index.html>。  
本指南已做成中英文对照阅读版的 pdf 文档，有兴趣的朋友可以去 [Java并发工具包java.util.concurrent用户指南中英文对照阅读版.pdf[带书签]](http://download.csdn.net/detail/defonds/8469189) 进行下载。

1. java.util.concurrent - Java 并发工具包

Java 5 添加了一个新的包到 Java 平台，java.util.concurrent 包。这个包包含有一系列能够让 Java 的并发编程变得更加简单轻松的类。在这个包被添加以前，你需要自己去动手实现自己的相关工具类。  
本文我将带你一一认识 java.util.concurrent 包里的这些类，然后你可以尝试着如何在项目中使用它们。本文中我将使用 Java 6 版本，我不确定这和 Java 5 版本里的是否有一些差异。  
我不会去解释关于 Java 并发的核心问题 - 其背后的原理，也就是说，如果你对那些东西感兴趣，参考《[Java 并发指南](http://tutorials.jenkov.com/java-concurrency/index.html)》。

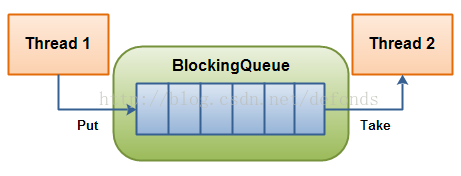
半成品

本文很大程度上还是个 "半成品"，所以当你发现一些被漏掉的类或接口时，请耐心等待。在我空闲的时候会把它们加进来的。

2. 阻塞队列 BlockingQueue

java.util.concurrent 包里的 BlockingQueue 接口表示一个线程安放入和提取实例的队列。本小节我将给你演示如何使用这个 BlockingQueue。  
本节不会讨论如何在 Java 中实现一个你自己的 BlockingQueue。如果你对那个感兴趣，参考《[Java 并发指南](http://tutorials.jenkov.com/java-concurrency/index.html)》

BlockingQueue 用法

BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景。下图是对这个原理的阐述：  
  
  
一个线程往里边放，另外一个线程从里边取的一个 BlockingQueue。  
一个线程将会持续生产新对象并将其插入到队列之中，直到队列达到它所能容纳的临界点。也就是说，它是有限的。如果该阻塞队列到达了其临界点，负责生产的线程将会在往里边插入新对象时发生阻塞。它会一直处于阻塞之中，直到负责消费的线程从队列中拿走一个对象。  
负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话，这个消费线程将会处于阻塞之中，直到一个生产线程把一个对象丢进队列。

BlockingQueue 的方法

BlockingQueue 具有 4 组不同的方法用于插入、移除以及对队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 抛异常 | 特定值 | 阻塞 | 超时 |
| 插入 | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| 移除 | remove(o) | poll(o) | take(o) | poll(timeout, timeunit) |
| 检查 | element(o) | peek(o) |  |  |

四组不同的行为方式解释：

抛异常：如果试图的操作无法立即执行，抛一个异常。

特定值：如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。

阻塞：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。

超时：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

无法向一个 BlockingQueue 中插入 null。如果你试图插入 null，BlockingQueue 将会抛出一个 NullPointerException。  
可以访问到 BlockingQueue 中的所有元素，而不仅仅是开始和结束的元素。比如说，你将一个对象放入队列之中以等待处理，但你的应用想要将其取消掉。那么你可以调用诸如 remove(o) 方法来将队列之中的特定对象进行移除。但是这么干效率并不高(译者注：基于队列的数据结构，获取除开始或结束位置的其他对象的效率不会太高)，因此你尽量不要用这一类的方法，除非你确实不得不那么做。

BlockingQueue 的实现

BlockingQueue 是个接口，你需要使用它的实现之一来使用 BlockingQueue。java.util.concurrent 具有以下 BlockingQueue 接口的实现(Java 6)：

[ArrayBlockingQueue](http://blog.csdn.net/defonds/article/details/44021605#t7)

[DelayQueue](http://blog.csdn.net/defonds/article/details/44021605#t8)

[LinkedBlockingQueue](http://blog.csdn.net/defonds/article/details/44021605#t9)

[PriorityBlockingQueue](http://blog.csdn.net/defonds/article/details/44021605#t10)

[SynchronousQueue](http://blog.csdn.net/defonds/article/details/44021605#t11)

Java 中使用 BlockingQueue 的例子

这里是一个 Java 中使用 BlockingQueue 的示例。本示例使用的是 BlockingQueue 接口的 ArrayBlockingQueue 实现。  
首先，BlockingQueueExample 类分别在两个独立的线程中启动了一个 Producer 和 一个 Consumer。Producer 向一个共享的 BlockingQueue 中注入字符串，而 Consumer 则会从中把它们拿出来。

public class BlockingQueueExample {

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

BlockingQueue queue = new ArrayBlockingQueue(1024);

Producer producer = new Producer(queue);

Consumer consumer = new Consumer(queue);

new Thread(producer).start();

new Thread(consumer).start();

Thread.sleep(4000);

}

}

以下是 Producer 类。注意它在每次 put() 调用时是如何休眠一秒钟的。这将导致 Consumer 在等待队列中对象的时候发生阻塞。

public class Producer implements Runnable{

protected BlockingQueue queue = null;

public Producer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

queue.put("1");

Thread.sleep(1000);

queue.put("2");

Thread.sleep(1000);

queue.put("3");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

以下是 Consumer 类。它只是把对象从队列中抽取出来，然后将它们打印到 System.out。

public class Consumer implements Runnable{

protected BlockingQueue queue = null;

public Consumer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

System.out.println(queue.take());

System.out.println(queue.take());

System.out.println(queue.take());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

3. 数组阻塞队列 ArrayBlockingQueue

ArrayBlockingQueue 类实现了 BlockingQueue 接口。  
ArrayBlockingQueue 是一个有界的阻塞队列，其内部实现是将对象放到一个数组里。有界也就意味着，它不能够存储无限多数量的元素。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(译者注：因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。  
ArrayBlockingQueue 内部以 FIFO(先进先出)的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。  
以下是在使用  ArrayBlockingQueue 的时候对其初始化的一个示例：

BlockingQueue queue = new ArrayBlockingQueue(1024);

queue.put("1");

Object object = queue.take();

以下是使用了 Java 泛型的一个 BlockingQueue 示例。注意其中是如何对 String 元素放入和提取的：

BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);

queue.put("1");

String string = queue.take();

4. 延迟队列 DelayQueue

DelayQueue 实现了 BlockingQueue 接口。  
DelayQueue 对元素进行持有直到一个特定的延迟到期。注入其中的元素必须实现 java.util.concurrent.Delayed 接口，该接口定义：

public interface Delayed extends Comparable<Delayed< {

public long getDelay(TimeUnit timeUnit);

}

DelayQueue 将会在每个元素的 getDelay() 方法返回的值的时间段之后才释放掉该元素。如果返回的是 0 或者负值，延迟将被认为过期，该元素将会在 DelayQueue 的下一次 take  被调用的时候被释放掉。  
传递给 getDelay 方法的 getDelay 实例是一个枚举类型，它表明了将要延迟的时间段。TimeUnit 枚举将会取以下值：

DAYS

HOURS

MINUTES

SECONDS

MILLISECONDS

MICROSECONDS

NANOSECONDS

正如你所看到的，Delayed 接口也继承了 java.lang.Comparable 接口，这也就意味着 Delayed 对象之间可以进行对比。这个可能在对 DelayQueue 队列中的元素进行排序时有用，因此它们可以根据过期时间进行有序释放。  
以下是使用 DelayQueue 的例子：

public class DelayQueueExample {

public static void main(String[] args) {

DelayQueue queue = new DelayQueue();

Delayed element1 = new DelayedElement();

queue.put(element1);

Delayed element2 = queue.take();

}

}

DelayedElement 是我所创建的一个 DelayedElement 接口的实现类，它不在 java.util.concurrent 包里。你需要自行创建你自己的 Delayed 接口的实现以使用 DelayQueue 类。

5. 链阻塞队列 LinkedBlockingQueue

LinkedBlockingQueue 类实现了 BlockingQueue 接口。  
LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer.MAX\_VALUE 作为上限。  
LinkedBlockingQueue 内部以 FIFO(先进先出)的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。  
以下是 LinkedBlockingQueue 的初始化和使用示例代码：

BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();

BlockingQueue<String> bounded = new LinkedBlockingQueue<String>(1024);

bounded.put("Value");

String value = bounded.take();

6. 具有优先级的阻塞队列 PriorityBlockingQueue

PriorityBlockingQueue 类实现了 BlockingQueue 接口。  
PriorityBlockingQueue 是一个无界的并发队列。它使用了和类 java.util.PriorityQueue 一样的排序规则。你无法向这个队列中插入 null 值。  
所有插入到 PriorityBlockingQueue 的元素必须实现 java.lang.Comparable 接口。因此该队列中元素的排序就取决于你自己的 Comparable 实现。  
注意 PriorityBlockingQueue 对于具有相等优先级(compare() == 0)的元素并不强制任何特定行为。  
同时注意，如果你从一个 PriorityBlockingQueue 获得一个 Iterator 的话，该 Iterator 并不能保证它对元素的遍历是以优先级为序的。  
以下是使用 PriorityBlockingQueue 的示例：

BlockingQueue queue = new PriorityBlockingQueue();

//String implements java.lang.Comparable

queue.put("Value");

String value = queue.take();

7. 同步队列 SynchronousQueue

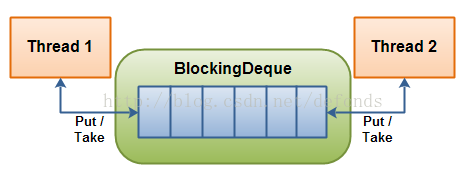
SynchronousQueue 类实现了 BlockingQueue 接口。  
SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。

据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。

8. 阻塞双端队列 BlockingDeque

java.util.concurrent 包里的 BlockingDeque 接口表示一个线程安放入和提取实例的双端队列。本小节我将给你演示如何使用 BlockingDeque。  
BlockingDeque 类是一个双端队列，在不能够插入元素时，它将阻塞住试图插入元素的线程；在不能够抽取元素时，它将阻塞住试图抽取的线程。  
deque(双端队列) 是 "Double Ended Queue" 的缩写。因此，双端队列是一个你可以从任意一端插入或者抽取元素的队列。

BlockingDeque 的使用

在线程既是一个队列的生产者又是这个队列的消费者的时候可以使用到 BlockingDeque。如果生产者线程需要在队列的两端都可以插入数据，消费者线程需要在队列的两端都可以移除数据，这个时候也可以使用 BlockingDeque。BlockingDeque 图解：  
  
  
一个 BlockingDeque - 线程在双端队列的两端都可以插入和提取元素。  
一个线程生产元素，并把它们插入到队列的任意一端。如果双端队列已满，插入线程将被阻塞，直到一个移除线程从该队列中移出了一个元素。如果双端队列为空，移除线程将被阻塞，直到一个插入线程向该队列插入了一个新元素。

BlockingDeque 的方法

BlockingDeque 具有 4 组不同的方法用于插入、移除以及对双端队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 抛异常 | 特定值 | 阻塞 | 超时 |
| 插入 | addFirst(o) | offerFirst(o) | putFirst(o) | offerFirst(o, timeout, timeunit) |
| 移除 | removeFirst(o) | pollFirst(o) | takeFirst(o) | pollFirst(timeout, timeunit) |
| 检查 | getFirst(o) | peekFirst(o) |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 抛异常 | 特定值 | 阻塞 | 超时 |
| 插入 | addLast(o) | offerLast(o) | putLast(o) | offerLast(o, timeout, timeunit) |
| 移除 | removeLast(o) | pollLast(o) | takeLast(o) | pollLast(timeout, timeunit) |
| 检查 | getLast(o) | peekLast(o) |  |  |

四组不同的行为方式解释：

抛异常：如果试图的操作无法立即执行，抛一个异常。

特定值：如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。

阻塞：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。

超时：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

BlockingDeque 继承自 BlockingQueue

BlockingDeque 接口继承自 BlockingQueue 接口。这就意味着你可以像使用一个 BlockingQueue 那样使用 BlockingDeque。如果你这么干的话，各种插入方法将会把新元素添加到双端队列的尾端，而移除方法将会把双端队列的首端的元素移除。正如 BlockingQueue 接口的插入和移除方法一样。  
以下是 BlockingDeque 对 BlockingQueue 接口的方法的具体内部实现：

|  |  |
| --- | --- |
| BlockingQueue | BlockingDeque |
| add() | addLast() |
| offer() x 2 | offerLast() x 2 |
| put() | putLast() |
|  |  |
| remove() | removeFirst() |
| poll() x 2 | pollFirst() |
| take() | takeFirst() |
|  |  |
| element() | getFirst() |
| peek() | peekFirst() |

BlockingDeque 的实现

既然 BlockingDeque 是一个接口，那么你想要使用它的话就得使用它的众多的实现类的其中一个。java.util.concurrent 包提供了以下 BlockingDeque 接口的实现类：

[LinkedBlockingDeque](http://blog.csdn.net/defonds/article/details/44021605#t18)

BlockingDeque 代码示例

以下是如何使用 BlockingDeque 方法的一个简短代码示例：

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

9. 链阻塞双端队列 LinkedBlockingDeque

LinkedBlockingDeque 类实现了 BlockingDeque 接口。  
deque(双端队列) 是 "Double Ended Queue" 的缩写。因此，双端队列是一个你可以从任意一端插入或者抽取元素的队列。(译者注：唐僧啊，受不了。)  
LinkedBlockingDeque 是一个双端队列，在它为空的时候，一个试图从中抽取数据的线程将会阻塞，无论该线程是试图从哪一端抽取数据。  
以下是 LinkedBlockingDeque 实例化以及使用的示例：

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

10. 并发 Map(映射) ConcurrentMap

java.util.concurrent.ConcurrentMap

java.util.concurrent.ConcurrentMap 接口表示了一个能够对别人的访问(插入和提取)进行并发处理的 java.util.Map。  
ConcurrentMap 除了从其父接口 java.util.Map 继承来的方法之外还有一些额外的原子性方法。

ConcurrentMap 的实现

既然 ConcurrentMap 是个接口，你想要使用它的话就得使用它的实现类之一。java.util.concurrent 包具备 ConcurrentMap 接口的以下实现类：

ConcurrentHashMap

ConcurrentHashMap

ConcurrentHashMap 和 java.util.HashTable 类很相似，但 ConcurrentHashMap 能够提供比 HashTable 更好的并发性能。在你从中读取对象的时候 ConcurrentHashMap 并不会把整个 Map 锁住。此外，在你向其中写入对象的时候，ConcurrentHashMap 也不会锁住整个 Map。它的内部只是把 Map 中正在被写入的部分进行锁定。  
另外一个不同点是，在被遍历的时候，即使是 ConcurrentHashMap 被改动，它也不会抛 ConcurrentModificationException。尽管 Iterator 的设计不是为多个线程的同时使用。  
更多关于 ConcurrentMap 和 ConcurrentHashMap 的细节请参考官方文档。

ConcurrentMap 例子

以下是如何使用 ConcurrentMap 接口的一个例子。本示例使用了 ConcurrentHashMap 实现类：

ConcurrentMap concurrentMap = new ConcurrentHashMap();

concurrentMap.put("key", "value");

Object value = concurrentMap.get("key");

11. 并发导航映射 ConcurrentNavigableMap

java.util.concurrent.ConcurrentNavigableMap 是一个支持并发访问的 java.util.NavigableMap，它还能让它的子 map 具备并发访问的能力。所谓的 "子 map" 指的是诸如 headMap()，subMap()，tailMap() 之类的方法返回的 map。

NavigableMap 中的方法不再赘述，本小节我们来看一下 ConcurrentNavigableMap 添加的方法。

headMap()

headMap(T toKey) 方法返回一个包含了小于给定 toKey 的 key 的子 map。  
如果你对原始 map 里的元素做了改动，这些改动将影响到子 map 中的元素(译者注：map 集合持有的其实只是对象的引用)。  
以下示例演示了对 headMap() 方法的使用：

ConcurrentNavigableMap map = new ConcurrentSkipListMap();

map.put("1", "one");

map.put("2", "two");

map.put("3", "three");

ConcurrentNavigableMap headMap = map.headMap("2");

headMap 将指向一个只含有键 "1" 的 ConcurrentNavigableMap，因为只有这一个键小于 "2"。关于这个方法及其重载版本具体是怎么工作的细节请参考 Java 文档。

tailMap()

tailMap(T fromKey) 方法返回一个包含了不小于给定 fromKey 的 key 的子 map。  
如果你对原始 map 里的元素做了改动，这些改动将影响到子 map 中的元素(译者注：map 集合持有的其实只是对象的引用)。  
以下示例演示了对 tailMap() 方法的使用：

ConcurrentNavigableMap map = new ConcurrentSkipListMap();

map.put("1", "one");

map.put("2", "two");

map.put("3", "three");

ConcurrentNavigableMap tailMap = map.tailMap("2");

tailMap 将拥有键 "2" 和 "3"，因为它们不小于给定键 "2"。关于这个方法及其重载版本具体是怎么工作的细节请参考 Java 文档。

subMap()

subMap() 方法返回原始 map 中，键介于 from(包含) 和 to (不包含) 之间的子 map。示例如下：

ConcurrentNavigableMap map = new ConcurrentSkipListMap();

map.put("1", "one");

map.put("2", "two");

map.put("3", "three");

ConcurrentNavigableMap subMap = map.subMap("2", "3");

返回的 submap 只包含键 "2"，因为只有它满足不小于 "2"，比 "3" 小。

更多方法

ConcurrentNavigableMap 接口还有其他一些方法可供使用，比如：

descendingKeySet()

descendingMap()

navigableKeySet()

关于这些方法更多信息参考官方 Java 文档。

12. 闭锁 CountDownLatch

java.util.concurrent.CountDownLatch 是一个并发构造，它允许一个或多个线程等待一系列指定操作的完成。  
CountDownLatch 以一个给定的数量初始化。countDown() 每被调用一次，这一数量就减一。通过调用 await() 方法之一，线程可以阻塞等待这一数量到达零。  
以下是一个简单示例。Decrementer 三次调用 countDown() 之后，等待中的 Waiter 才会从 await() 调用中释放出来。

CountDownLatch latch = new CountDownLatch(3);

Waiter waiter = new Waiter(latch);

Decrementer decrementer = new Decrementer(latch);

new Thread(waiter) .start();

new Thread(decrementer).start();

Thread.sleep(4000);

public class Waiter implements Runnable{

CountDownLatch latch = null;

public Waiter(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

try {

latch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Waiter Released");

}

}

public class Decrementer implements Runnable {

CountDownLatch latch = null;

public Decrementer(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

try {

Thread.sleep(1000);

this.latch.countDown();

Thread.sleep(1000);

this.latch.countDown();

Thread.sleep(1000);

this.latch.countDown();

} catch (InterruptedException e) {

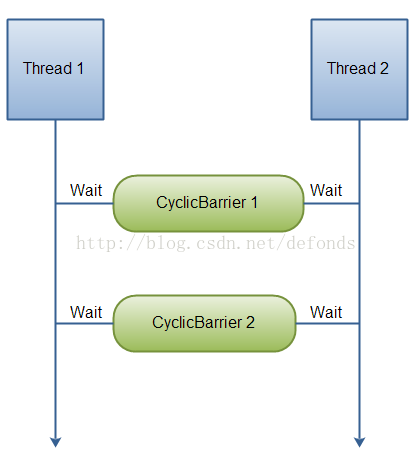
e.printStackTrace();

}

}

}

13. 栅栏 CyclicBarrier

java.util.concurrent.CyclicBarrier 类是一种同步机制，它能够对处理一些算法的线程实现同步。换句话讲，它就是一个所有线程必须等待的一个栅栏，直到所有线程都到达这里，然后所有线程才可以继续做其他事情。图示如下：  
  
  
两个线程在栅栏旁等待对方。  
通过调用 CyclicBarrier 对象的 await() 方法，两个线程可以实现互相等待。一旦 N 个线程在等待 CyclicBarrier 达成，所有线程将被释放掉去继续运行。

创建一个 CyclicBarrier

在创建一个 CyclicBarrier 的时候你需要定义有多少线程在被释放之前等待栅栏。创建 CyclicBarrier 示例：

CyclicBarrier barrier = new CyclicBarrier(2);

等待一个 CyclicBarrier

以下演示了如何让一个线程等待一个 CyclicBarrier：

barrier.await();

当然，你也可以为等待线程设定一个超时时间。等待超过了超时时间之后，即便还没有达成 N 个线程等待 CyclicBarrier 的条件，该线程也会被释放出来。以下是定义超时时间示例：

barrier.await(10, TimeUnit.SECONDS);

满足以下任何条件都可以让等待 CyclicBarrier 的线程释放：

最后一个线程也到达 CyclicBarrier(调用 await())

当前线程被其他线程打断(其他线程调用了这个线程的 interrupt() 方法)

其他等待栅栏的线程被打断

其他等待栅栏的线程因超时而被释放

外部线程调用了栅栏的 CyclicBarrier.reset() 方法

CyclicBarrier 行动

CyclicBarrier 支持一个栅栏行动，栅栏行动是一个 Runnable 实例，一旦最后等待栅栏的线程抵达，该实例将被执行。你可以在 CyclicBarrier 的构造方法中将 Runnable 栅栏行动传给它：

Runnable barrierAction = ... ;

CyclicBarrier barrier = new CyclicBarrier(2, barrierAction);

CyclicBarrier 示例

以下代码演示了如何使用 CyclicBarrier：

Runnable barrier1Action = new Runnable() {

public void run() {

System.out.println("BarrierAction 1 executed ");

}

};

Runnable barrier2Action = new Runnable() {

public void run() {

System.out.println("BarrierAction 2 executed ");

}

};

CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);

CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);

CyclicBarrierRunnable barrierRunnable1 =

new CyclicBarrierRunnable(barrier1, barrier2);

CyclicBarrierRunnable barrierRunnable2 =

new CyclicBarrierRunnable(barrier1, barrier2);

new Thread(barrierRunnable1).start();

new Thread(barrierRunnable2).start();

CyclicBarrierRunnable 类：

public class CyclicBarrierRunnable implements Runnable{

CyclicBarrier barrier1 = null;

CyclicBarrier barrier2 = null;

public CyclicBarrierRunnable(

CyclicBarrier barrier1,

CyclicBarrier barrier2) {

this.barrier1 = barrier1;

this.barrier2 = barrier2;

}

public void run() {

try {

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() +

" waiting at barrier 1");

this.barrier1.await();

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() +

" waiting at barrier 2");

this.barrier2.await();

System.out.println(Thread.currentThread().getName() +

" done!");

} catch (InterruptedException e) {

e.printStackTrace();

} catch (BrokenBarrierException e) {

e.printStackTrace();

}

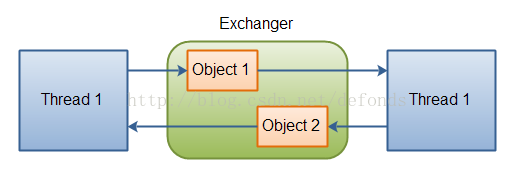
}

}

以上代码控制台输出如下。注意每个线程写入控制台的时序可能会跟你实际执行不一样。比如有时 Thread-0 先打印，有时 Thread-1 先打印。  
Thread-0 waiting at barrier 1  
Thread-1 waiting at barrier 1  
BarrierAction 1 executed  
Thread-1 waiting at barrier 2  
Thread-0 waiting at barrier 2  
BarrierAction 2 executed  
Thread-0 done!

Thread-1 done!

14. 交换机 Exchanger

java.util.concurrent.Exchanger 类表示一种两个线程可以进行互相交换对象的会和点。这种机制图示如下：  
  
  
两个线程通过一个 Exchanger 交换对象。  
交换对象的动作由 Exchanger 的两个 exchange() 方法的其中一个完成。以下是一个示例：

Exchanger exchanger = new Exchanger();

ExchangerRunnable exchangerRunnable1 =

new ExchangerRunnable(exchanger, "A");

ExchangerRunnable exchangerRunnable2 =

new ExchangerRunnable(exchanger, "B");

new Thread(exchangerRunnable1).start();

new Thread(exchangerRunnable2).start();

ExchangerRunnable 代码：

public class ExchangerRunnable implements Runnable{

Exchanger exchanger = null;

Object object = null;

public ExchangerRunnable(Exchanger exchanger, Object object) {

this.exchanger = exchanger;

this.object = object;

}

public void run() {

try {

Object previous = this.object;

this.object = this.exchanger.exchange(this.object);

System.out.println(

Thread.currentThread().getName() +

" exchanged " + previous + " for " + this.object

);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

以上程序输出：  
Thread-0 exchanged A for B

Thread-1 exchanged B for A

15. 信号量 Semaphore

java.util.concurrent.Semaphore 类是一个计数信号量。这就意味着它具备两个主要方法：

acquire()

release()

计数信号量由一个指定数量的 "许可" 初始化。每调用一次 acquire()，一个许可会被调用线程取走。每调用一次 release()，一个许可会被返还给信号量。因此，在没有任何 release() 调用时，最多有 N 个线程能够通过 acquire() 方法，N 是该信号量初始化时的许可的指定数量。这些许可只是一个简单的计数器。这里没啥奇特的地方。

Semaphore 用法

信号量主要有两种用途：

保护一个重要(代码)部分防止一次超过 N 个线程进入。

在两个线程之间发送信号。

保护重要部分

如果你将信号量用于保护一个重要部分，试图进入这一部分的代码通常会首先尝试获得一个许可，然后才能进入重要部分(代码块)，执行完之后，再把许可释放掉。比如这样：

Semaphore semaphore = new Semaphore(1);

//critical section

semaphore.acquire();

...

semaphore.release();

在线程之间发送信号

如果你将一个信号量用于在两个线程之间传送信号，通常你应该用一个线程调用 acquire() 方法，而另一个线程调用 release() 方法。  
如果没有可用的许可，acquire() 调用将会阻塞，直到一个许可被另一个线程释放出来。同理，如果无法往信号量释放更多许可时，一个 release() 调用也会阻塞。  
通过这个可以对多个线程进行协调。比如，如果线程 1 将一个对象插入到了一个共享列表(list)之后之后调用了 acquire()，而线程 2 则在从该列表中获取一个对象之前调用了 release()，这时你其实已经创建了一个阻塞队列。信号量中可用的许可的数量也就等同于该阻塞队列能够持有的元素个数。

公平

没有办法保证线程能够公平地可从信号量中获得许可。也就是说，无法担保掉第一个调用 acquire() 的线程会是第一个获得一个许可的线程。如果第一个线程在等待一个许可时发生阻塞，而第二个线程前来索要一个许可的时候刚好有一个许可被释放出来，那么它就可能会在第一个线程之前获得许可。  
如果你想要强制公平，Semaphore 类有一个具有一个布尔类型的参数的构造子，通过这个参数以告知 Semaphore 是否要强制公平。强制公平会影响到并发性能，所以除非你确实需要它否则不要启用它。  
以下是如何在公平模式创建一个 Semaphore 的示例：

Semaphore semaphore = new Semaphore(1, true);

更多方法

java.util.concurrent.Semaphore 类还有很多方法，比如：

availablePermits()

acquireUninterruptibly()

drainPermits()

hasQueuedThreads()

getQueuedThreads()

tryAcquire()

等等

这些方法的细节请参考 Java 文档。

16. 执行器服务 ExecutorService

java.util.concurrent.ExecutorService 接口表示一个异步执行机制，使我们能够在后台执行任务。因此一个 ExecutorService 很类似于一个线程池。实际上，存在于 java.util.concurrent 包里的 ExecutorService 实现就是一个线程池实现。

ExecutorService 例子

以下是一个简单的 ExecutorService 例子：

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

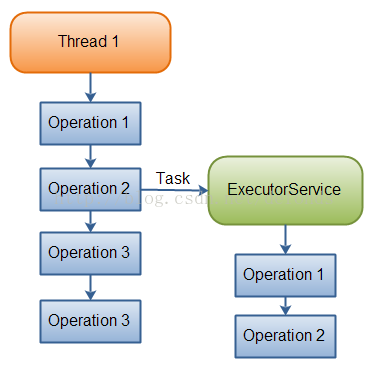
}

});

executorService.shutdown();

首先使用 newFixedThreadPool() 工厂方法创建一个 ExecutorService。这里创建了一个十个线程执行任务的线程池。  
然后，将一个 Runnable 接口的匿名实现类传递给 execute() 方法。这将导致 ExecutorService 中的某个线程执行该 Runnable。

任务委派

下图说明了一个线程是如何将一个任务委托给一个 ExecutorService 去异步执行的：  
  
  
一个线程将一个任务委派给一个 ExecutorService 去异步执行。  
一旦该线程将任务委派给 ExecutorService，该线程将继续它自己的执行，独立于该任务的执行。

ExecutorService 实现

既然 ExecutorService 是个接口，如果你想用它的话就得去使用它的实现类之一。java.util.concurrent 包提供了 ExecutorService 接口的以下实现类：

[ThreadPoolExecutor](http://blog.csdn.net/defonds/article/details/44021605#t53)

[ScheduledThreadPoolExecutor](http://blog.csdn.net/defonds/article/details/44021605#t55)

创建一个 ExecutorService

ExecutorService 的创建依赖于你使用的具体实现。但是你也可以使用 Executors 工厂类来创建 ExecutorService 实例。以下是几个创建 ExecutorService 实例的例子：

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

ExecutorService 使用

有几种不同的方式来将任务委托给 ExecutorService 去执行：

execute(Runnable)

submit(Runnable)

submit(Callable)

invokeAny(...)

invokeAll(...)

接下来我们挨个看一下这些方法。

execute(Runnable)

execute(Runnable) 方法要求一个 java.lang.Runnable 对象，然后对它进行异步执行。以下是使用 ExecutorService 执行一个 Runnable 的示例：

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

没有办法得知被执行的 Runnable 的执行结果。如果有需要的话你得使用一个 Callable(以下将做介绍)。

submit(Runnable)

submit(Runnable) 方法也要求一个 Runnable 实现类，但它返回一个 Future 对象。这个 Future 对象可以用来检查 Runnable 是否已经执行完毕。  
以下是 ExecutorService submit() 示例：

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

submit(Callable)

submit(Callable) 方法类似于 submit(Runnable) 方法，除了它所要求的参数类型之外。Callable 实例除了它的 call() 方法能够返回一个结果之外和一个 Runnable 很相像。Runnable.run() 不能够返回一个结果。  
Callable 的结果可以通过 submit(Callable) 方法返回的 Future 对象进行获取。以下是一个 ExecutorService Callable 示例：

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

以上代码输出：  
Asynchronous Callable  
future.get() = Callable Result

invokeAny()

invokeAny() 方法要求一系列的 Callable 或者其子接口的实例对象。调用这个方法并不会返回一个 Future，但它返回其中一个 Callable 对象的结果。无法保证返回的是哪个 Callable 的结果 - 只能表明其中一个已执行结束。  
如果其中一个任务执行结束(或者抛了一个异常)，其他 Callable 将被取消。  
以下是示例代码：

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

上述代码将会打印出给定 Callable 集合中的一个的执行结果。我自己试着执行了它几次，结果始终在变。有时是 "Task 1"，有时是 "Task 2" 等等。

invokeAll()

invokeAll() 方法将调用你在集合中传给 ExecutorService 的所有 Callable 对象。invokeAll() 返回一系列的 Future 对象，通过它们你可以获取每个 Callable 的执行结果。  
记住，一个任务可能会由于一个异常而结束，因此它可能没有 "成功"。无法通过一个 Future 对象来告知我们是两种结束中的哪一种。  
以下是一个代码示例：

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

ExecutorService 关闭

使用完 ExecutorService 之后你应该将其关闭，以使其中的线程不再运行。  
比如，如果你的应用是通过一个 main() 方法启动的，之后 main 方法退出了你的应用，如果你的应用有一个活动的 ExexutorService 它将还会保持运行。ExecutorService 里的活动线程阻止了 JVM 的关闭。  
要终止 ExecutorService 里的线程你需要调用 ExecutorService 的 shutdown() 方法。ExecutorService 并不会立即关闭，但它将不再接受新的任务，而且一旦所有线程都完成了当前任务的时候，ExecutorService 将会关闭。在 shutdown() 被调用之前所有提交给 ExecutorService 的任务都被执行。

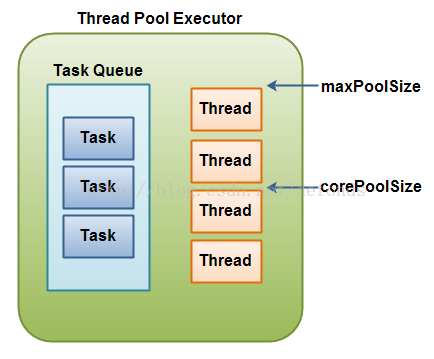
如果你想要立即关闭 ExecutorService，你可以调用 shutdownNow() 方法。这样会立即尝试停止所有执行中的任务，并忽略掉那些已提交但尚未开始处理的任务。无法担保执行任务的正确执行。可能它们被停止了，也可能已经执行结束。

17. 线程池执行者 ThreadPoolExecutor

java.util.concurrent.ThreadPoolExecutor 是 ExecutorService 接口的一个实现。ThreadPoolExecutor 使用其内部池中的线程执行给定任务(Callable 或者 Runnable)。  
ThreadPoolExecutor 包含的线程池能够包含不同数量的线程。池中线程的数量由以下变量决定：

corePoolSize

maximumPoolSize

当一个任务委托给线程池时，如果池中线程数量低于 corePoolSize，一个新的线程将被创建，即使池中可能尚有空闲线程。  
如果内部任务队列已满，而且有至少 corePoolSize 正在运行，但是运行线程的数量低于 maximumPoolSize，一个新的线程将被创建去执行该任务。  
ThreadPoolExecutor 图解：  
  
  
一个 ThreadPoolExecutor

创建一个 ThreadPoolExecutor

ThreadPoolExecutor 有若干个可用构造子。比如：

int corePoolSize = 5;

int maxPoolSize = 10;

long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =

new ThreadPoolExecutor(

corePoolSize,

maxPoolSize,

keepAliveTime,

TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()

);

但是，除非你确实需要显式为 ThreadPoolExecutor 定义所有参数，使用 java.util.concurrent.Executors 类中的工厂方法之一会更加方便，正如  [ExecutorService](http://blog.csdn.net/defonds/article/details/44021605#t41) 小节所述。

18.  定时执行者服务 ScheduledExecutorService

java.util.concurrent.ScheduledExecutorService 是一个 ExecutorService， 它能够将任务延后执行，或者间隔固定时间多次执行。 任务由一个工作者线程异步执行，而不是由提交任务给 ScheduledExecutorService 的那个线程执行。

ScheduledExecutorService 例子

以下是一个简单的 ScheduledExecutorService 示例：

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

首先一个内置 5 个线程的 ScheduledExecutorService 被创建。之后一个 Callable 接口的匿名类示例被创建然后传递给 schedule() 方法。后边的俩参数定义了 Callable 将在 5 秒钟之后被执行。

ScheduledExecutorService 实现

既然 ScheduledExecutorService 是一个接口，你要用它的话就得使用 java.util.concurrent 包里对它的某个实现类。ScheduledExecutorService 具有以下实现类：

ScheduledThreadPoolExecutor

创建一个 ScheduledExecutorService

如何创建一个 ScheduledExecutorService 取决于你采用的它的实现类。但是你也可以使用 Executors 工厂类来创建一个 ScheduledExecutorService 实例。比如：

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledExecutorService 使用

一旦你创建了一个 ScheduledExecutorService，你可以通过调用它的以下方法：

schedule (Callable task, long delay, TimeUnit timeunit)

schedule (Runnable task, long delay, TimeUnit timeunit)

scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)

scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

下面我们就简单看一下这些方法。

schedule (Callable task, long delay, TimeUnit timeunit)

这个方法计划指定的 Callable 在给定的延迟之后执行。  
这个方法返回一个 ScheduledFuture，通过它你可以在它被执行之前对它进行取消，或者在它执行之后获取结果。  
以下是一个示例：

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

System.out.println("result = " + scheduledFuture.get());

scheduledExecutorService.shutdown();

示例输出结果：  
Executed!  
result = Called!

schedule (Runnable task, long delay, TimeUnit timeunit)

除了 Runnable 无法返回一个结果之外，这一方法工作起来就像以一个 Callable 作为一个参数的那个版本的方法一样，因此 ScheduledFuture.get() 在任务执行结束之后返回 null。

scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)

这一方法规划一个任务将被定期执行。该任务将会在首个 initialDelay 之后得到执行，然后每个 period 时间之后重复执行。  
如果给定任务的执行抛出了异常，该任务将不再执行。如果没有任何异常的话，这个任务将会持续循环执行到 ScheduledExecutorService 被关闭。  
如果一个任务占用了比计划的时间间隔更长的时候，下一次执行将在当前执行结束执行才开始。计划任务在同一时间不会有多个线程同时执行。

scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

除了 period 有不同的解释之外这个方法和 scheduleAtFixedRate() 非常像。  
scheduleAtFixedRate() 方法中，period 被解释为前一个执行的开始和下一个执行的开始之间的间隔时间。  
而在本方法中，period 则被解释为前一个执行的结束和下一个执行的结束之间的间隔。因此这个延迟是执行结束之间的间隔，而不是执行开始之间的间隔。

ScheduledExecutorService 关闭

正如 ExecutorService，在你使用结束之后你需要把 ScheduledExecutorService 关闭掉。否则他将导致 JVM 继续运行，即使所有其他线程已经全被关闭。

你可以使用从 ExecutorService 接口继承来的 shutdown() 或 shutdownNow() 方法将 ScheduledExecutorService 关闭。参见 ExecutorService 关闭部分以获取更多信息。

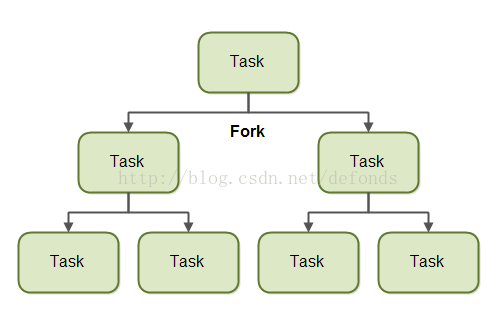
19. 使用 ForkJoinPool 进行分叉和合并

ForkJoinPool 在 Java 7 中被引入。它和 [ExecutorService](http://blog.csdn.net/defonds/article/details/44021605#t41) 很相似，除了一点不同。ForkJoinPool 让我们可以很方便地把任务分裂成几个更小的任务，这些分裂出来的任务也将会提交给 ForkJoinPool。任务可以继续分割成更小的子任务，只要它还能分割。可能听起来有些抽象，因此本节中我们将会解释 ForkJoinPool 是如何工作的，还有任务分割是如何进行的。

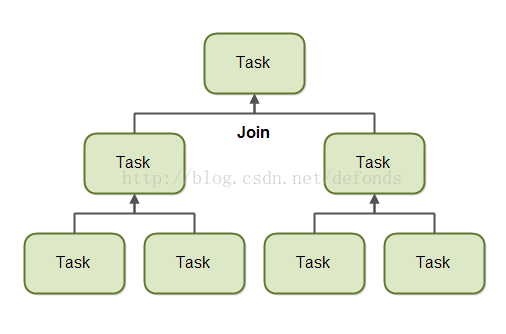
分叉和合并解释

在我们开始看 ForkJoinPool 之前我们先来简要解释一下分叉和合并的原理。  
分叉和合并原理包含两个递归进行的步骤。两个步骤分别是分叉步骤和合并步骤。

分叉

一个使用了分叉和合并原理的任务可以将自己分叉(分割)为更小的子任务，这些子任务可以被并发执行。如下图所示：  
  
通过把自己分割成多个子任务，每个子任务可以由不同的 CPU 并行执行，或者被同一个 CPU 上的不同线程执行。  
只有当给的任务过大，把它分割成几个子任务才有意义。把任务分割成子任务有一定开销，因此对于小型任务，这个分割的消耗可能比每个子任务并发执行的消耗还要大。  
什么时候把一个任务分割成子任务是有意义的，这个界限也称作一个阀值。这要看每个任务对有意义阀值的决定。很大程度上取决于它要做的工作的种类。

合并

当一个任务将自己分割成若干子任务之后，该任务将进入等待所有子任务的结束之中。  
一旦子任务执行结束，该任务可以把所有结果合并到同一个结果。图示如下：  
  
当然，并非所有类型的任务都会返回一个结果。如果这个任务并不返回一个结果，它只需等待所有子任务执行完毕。也就不需要结果的合并啦。

ForkJoinPool

ForkJoinPool 是一个特殊的线程池，它的设计是为了更好的配合 分叉-和-合并 任务分割的工作。ForkJoinPool 也在 java.util.concurrent 包中，其完整类名为 java.util.concurrent.ForkJoinPool。

创建一个 ForkJoinPool

你可以通过其构造子创建一个 ForkJoinPool。作为传递给 ForkJoinPool 构造子的一个参数，你可以定义你期望的并行级别。并行级别表示你想要传递给 ForkJoinPool 的任务所需的线程或 CPU 数量。以下是一个 ForkJoinPool 示例：

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

这个示例创建了一个并行级别为 4 的 ForkJoinPool。

提交任务到 ForkJoinPool

就像提交任务到 ExecutorService 那样，把任务提交到 ForkJoinPool。你可以提交两种类型的任务。一种是没有任何返回值的(一个 "行动")，另一种是有返回值的(一个"任务")。这两种类型分别由 RecursiveAction 和 RecursiveTask 表示。接下来介绍如何使用这两种类型的任务，以及如何对它们进行提交。

RecursiveAction

RecursiveAction 是一种没有任何返回值的任务。它只是做一些工作，比如写数据到磁盘，然后就退出了。  
一个 RecursiveAction 可以把自己的工作分割成更小的几块，这样它们可以由独立的线程或者 CPU 执行。  
你可以通过继承来实现一个 RecursiveAction。示例如下：

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

private long workLoad = 0;

public MyRecursiveAction(long workLoad) {

this.workLoad = workLoad;

}

@Override

protected void compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveAction> subtasks =

new ArrayList<MyRecursiveAction>();

subtasks.addAll(createSubtasks());

for(RecursiveAction subtask : subtasks){

subtask.fork();

}

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

}

}

private List<MyRecursiveAction> createSubtasks() {

List<MyRecursiveAction> subtasks =

new ArrayList<MyRecursiveAction>();

MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad / 2);

MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

例子很简单。MyRecursiveAction 将一个虚构的 workLoad 作为参数传给自己的构造子。如果 workLoad 高于一个特定阀值，该工作将被分割为几个子工作，子工作继续分割。如果 workLoad 低于特定阀值，该工作将由 MyRecursiveAction 自己执行。  
你可以这样规划一个 MyRecursiveAction  的执行：

MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);

forkJoinPool.invoke(myRecursiveAction);

RecursiveTask

RecursiveTask 是一种会返回结果的任务。它可以将自己的工作分割为若干更小任务，并将这些子任务的执行结果合并到一个集体结果。可以有几个水平的分割和合并。以下是一个 RecursiveTask 示例：

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Long> {

private long workLoad = 0;

public MyRecursiveTask(long workLoad) {

this.workLoad = workLoad;

}

protected Long compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

subtasks.addAll(createSubtasks());

for(MyRecursiveTask subtask : subtasks){

subtask.fork();

}

long result = 0;

for(MyRecursiveTask subtask : subtasks) {

result += subtask.join();

}

return result;

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

return workLoad \* 3;

}

}

private List<MyRecursiveTask> createSubtasks() {

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);

MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

除了有一个结果返回之外，这个示例和 RecursiveAction  的例子很像。MyRecursiveTask 类继承自 RecursiveTask<Long>，这也就意味着它将返回一个 Long 类型的结果。  
MyRecursiveTask 示例也会将工作分割为子任务，并通过 fork() 方法对这些子任务计划执行。  
此外，本示例还通过调用每个子任务的 join() 方法收集它们返回的结果。子任务的结果随后被合并到一个更大的结果，并最终将其返回。对于不同级别的递归，这种子任务的结果合并可能会发生递归。  
你可以这样规划一个 RecursiveTask：

MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);

注意是如何通过 ForkJoinPool.invoke() 方法的调用来获取最终执行结果的。

ForkJoinPool 评论

貌似并非每个人都对 Java 7 里的 ForkJoinPool 满意：《[一个 Java 分叉-合并 带来的灾祸](http://coopsoft.com/ar/CalamityArticle.html)》。

在你计划在自己的项目里使用 ForkJoinPool 之前最好读一下该篇文章。

20. 锁 Lock

java.util.concurrent.locks.Lock 是一个类似于 synchronized 块的线程同步机制。但是 Lock 比 synchronized 块更加灵活、精细。  
顺便说一下，在我的《[Java 并发指南](http://tutorials.jenkov.com/java-concurrency/index.html)》中我对如何实现你自己的锁进行了描述。

Java Lock 例子

既然 Lock 是一个接口，在你的程序里需要使用它的实现类之一来使用它。以下是一个简单示例：

Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();

首先创建了一个 Lock 对象。之后调用了它的 lock() 方法。这时候这个 lock 实例就被锁住啦。任何其他再过来调用 lock() 方法的线程将会被阻塞住，直到锁定 lock 实例的线程调用了 unlock() 方法。最后 unlock() 被调用了，lock 对象解锁了，其他线程可以对它进行锁定了。

Java Lock 实现

java.util.concurrent.locks 包提供了以下对 Lock 接口的实现类：

ReentrantLock

Lock 和 synchronized 代码块的主要不同点

一个 Lock 对象和一个 synchronized 代码块之间的主要不同点是：

synchronized 代码块不能够保证进入访问等待的线程的先后顺序。

你不能够传递任何参数给一个 synchronized 代码块的入口。因此，对于 synchronized 代码块的访问等待设置超时时间是不可能的事情。

synchronized 块必须被完整地包含在单个方法里。而一个 Lock 对象可以把它的 lock() 和 unlock() 方法的调用放在不同的方法里。

Lock 的方法

Lock 接口具有以下主要方法：

lock()

lockInterruptibly()

tryLock()

tryLock(long timeout, TimeUnit timeUnit)

unlock()

lock() 将 Lock 实例锁定。如果该 Lock 实例已被锁定，调用 lock() 方法的线程将会阻塞，直到 Lock 实例解锁。  
lockInterruptibly() 方法将会被调用线程锁定，除非该线程被打断。此外，如果一个线程在通过这个方法来锁定 Lock 对象时进入阻塞等待，而它被打断了的话，该线程将会退出这个方法调用。  
tryLock() 方法试图立即锁定 Lock 实例。如果锁定成功，它将返回 true，如果 Lock 实例已被锁定该方法返回 false。这一方法永不阻塞。  
tryLock(long timeout, TimeUnit timeUnit) 的工作类似于 tryLock() 方法，除了它在放弃锁定 Lock 之前等待一个给定的超时时间之外。

unlock() 方法对 Lock 实例解锁。一个 Lock 实现将只允许锁定了该对象的线程来调用此方法。其他(没有锁定该 Lock 对象的线程)线程对 unlock() 方法的调用将会抛一个未检查异常(RuntimeException)。

21. 读写锁 ReadWriteLock

java.util.concurrent.locks.ReadWriteLock 读写锁是一种先进的线程锁机制。它能够允许多个线程在同一时间对某特定资源进行读取，但同一时间内只能有一个线程对其进行写入。  
读写锁的理念在于多个线程能够对一个共享资源进行读取，而不会导致并发问题。并发问题的发生场景在于对一个共享资源的读和写操作的同时进行，或者多个写操作并发进行。  
本节只讨论 Java 内置 ReadWriteLock。如果你想了解 ReadWriteLock 背后的实现原理，请参考我的《Java 并发指南》主题中的《[读写锁](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html)》小节。

ReadWriteLock 锁规则

一个线程在对受保护资源在读或者写之前对 ReadWriteLock 锁定的规则如下：

读锁：如果没有任何写操作线程锁定 ReadWriteLock，并且没有任何写操作线程要求一个写锁(但还没有获得该锁)。因此，可以有多个读操作线程对该锁进行锁定。

写锁：如果没有任何读操作或者写操作。因此，在写操作的时候，只能有一个线程对该锁进行锁定。

ReadWriteLock 实现

ReadWriteLock 是个接口，如果你想用它的话就得去使用它的实现类之一。java.util.concurrent.locks 包提供了 ReadWriteLock 接口的以下实现类：

ReentrantReadWriteLock

ReadWriteLock 代码示例

以下是 ReadWriteLock 的创建以及如何使用它进行读、写锁定的简单示例代码：

ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();

// multiple readers can enter this section

// if not locked for writing, and not writers waiting

// to lock for writing.

readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();

// only one writer can enter this section,

// and only if no threads are currently reading.

readWriteLock.writeLock().unlock();

注意如何使用 ReadWriteLock 对两种锁实例的持有。一个对读访问进行保护，一个队写访问进行保护。

22. 原子性布尔 AtomicBoolean

AtomicBoolean 类为我们提供了一个可以用原子方式进行读和写的布尔值，它还拥有一些先进的原子性操作，比如 compareAndSet()。AtomicBoolean 类位于 java.util.concurrent.atomic 包，完整类名是为 java.util.concurrent.atomic.AtomicBoolean。本小节描述的 AtomicBoolean 是 Java 8 版本里的，而不是它第一次被引入的 Java 5 版本。  
AtomicBoolean 背后的设计理念在我的《Java 并发指南》主题的《[比较和交换](http://tutorials.jenkov.com/java-concurrency/compare-and-swap.html)》小节有解释。

创建一个 AtomicBoolean

你可以这样创建一个 AtomicBoolean：

AtomicBoolean atomicBoolean = new AtomicBoolean();

以上示例新建了一个默认值为 false 的 AtomicBoolean。  
如果你想要为 AtomicBoolean 实例设置一个显式的初始值，那么你可以将初始值传给 AtomicBoolean 的构造子：

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

获取 AtomicBoolean 的值

你可以通过使用 get() 方法来获取一个 AtomicBoolean 的值。示例如下：

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean value = atomicBoolean.get();

以上代码执行后 value 变量的值将为 true。

设置 AtomicBoolean 的值

你可以通过使用 set() 方法来设置一个 AtomicBoolean 的值。示例如下：

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

atomicBoolean.set(false);

以上代码执行后 AtomicBoolean 的值为 false。

交换 AtomicBoolean 的值

你可以通过 getAndSet() 方法来交换一个 AtomicBoolean 实例的值。getAndSet() 方法将返回 AtomicBoolean 当前的值，并将为 AtomicBoolean 设置一个新值。示例如下：

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean oldValue = atomicBoolean.getAndSet(false);

以上代码执行后 oldValue 变量的值为 true，atomicBoolean 实例将持有 false 值。代码成功将 AtomicBoolean 当前值 ture 交换为 false。

比较并设置 AtomicBoolean 的值

compareAndSet() 方法允许你对 AtomicBoolean 的当前值与一个期望值进行比较，如果当前值等于期望值的话，将会对 AtomicBoolean 设定一个新值。compareAndSet() 方法是原子性的，因此在同一时间之内有单个线程执行它。因此 compareAndSet() 方法可被用于一些类似于锁的同步的简单实现。  
以下是一个 compareAndSet() 示例：

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean expectedValue = true;

boolean newValue = false;

boolean wasNewValueSet = atomicBoolean.compareAndSet(

expectedValue, newValue);

本示例对 AtomicBoolean 的当前值与 true 值进行比较，如果相等，将 AtomicBoolean 的值更新为 false。

23. 原子性整型 AtomicInteger

AtomicInteger 类为我们提供了一个可以进行原子性读和写操作的 int 变量，它还包含一系列先进的原子性操作，比如 compareAndSet()。AtomicInteger 类位于 java.util.concurrent.atomic 包，因此其完整类名为 java.util.concurrent.atomic.AtomicInteger。本小节描述的 AtomicInteger 是 Java 8 版本里的，而不是它第一次被引入的 Java 5 版本。  
AtomicInteger 背后的设计理念在我的《Java 并发指南》主题的《[比较和交换](http://tutorials.jenkov.com/java-concurrency/compare-and-swap.html)》小节有解释。

创建一个 AtomicInteger

创建一个 AtomicInteger 示例如下：

AtomicInteger atomicInteger = new AtomicInteger();

本示例将创建一个初始值为 0 的 AtomicInteger。  
如果你想要创建一个给定初始值的 AtomicInteger，你可以这样：

AtomicInteger atomicInteger = new AtomicInteger(123);

本示例将 123 作为参数传给 AtomicInteger 的构造子，它将设置 AtomicInteger 实例的初始值为 123。

获取 AtomicInteger 的值

你可以使用 get() 方法获取 AtomicInteger 实例的值。示例如下：

AtomicInteger atomicInteger = new AtomicInteger(123);

int theValue = atomicInteger.get();

设置 AtomicInteger 的值

你可以通过 set() 方法对 AtomicInteger 的值进行重新设置。以下是 AtomicInteger.set() 示例：

AtomicInteger atomicInteger = new AtomicInteger(123);

atomicInteger.set(234);

以上示例创建了一个初始值为 123 的 AtomicInteger，而在第二行将其值更新为 234。

比较并设置 AtomicInteger 的值

AtomicInteger 类也通过了一个原子性的 compareAndSet() 方法。这一方法将 AtomicInteger 实例的当前值与期望值进行比较，如果二者相等，为 AtomicInteger 实例设置一个新值。AtomicInteger.compareAndSet() 代码示例：

AtomicInteger atomicInteger = new AtomicInteger(123);

int expectedValue = 123;

int newValue = 234;

atomicInteger.compareAndSet(expectedValue, newValue);

本示例首先新建一个初始值为 123 的 AtomicInteger 实例。然后将 AtomicInteger 与期望值 123 进行比较，如果相等，将 AtomicInteger 的值更新为 234。

增加 AtomicInteger 值

AtomicInteger 类包含有一些方法，通过它们你可以增加 AtomicInteger 的值，并获取其值。这些方法如下：

addAndGet()

getAndAdd()

getAndIncrement()

incrementAndGet()

第一个 addAndGet() 方法给 AtomicInteger 增加了一个值，然后返回增加后的值。getAndAdd() 方法为 AtomicInteger 增加了一个值，但返回的是增加以前的 AtomicInteger 的值。具体使用哪一个取决于你的应用场景。以下是这两种方法的示例：

AtomicInteger atomicInteger = new AtomicInteger();

System.out.println(atomicInteger.getAndAdd(10));

System.out.println(atomicInteger.addAndGet(10));

本示例将打印出 0 和 20。例子中，第二行拿到的是加 10 之前的 AtomicInteger 的值。加 10 之前的值是 0。第三行将 AtomicInteger 的值再加 10，并返回加操作之后的值。该值现在是为 20。  
你当然也可以使用这俩方法为 AtomicInteger  添加负值。结果实际是一个减法操作。  
getAndIncrement() 和 incrementAndGet() 方法类似于 getAndAdd() 和 addAndGet()，但每次只将 AtomicInteger 的值加 1。

减小 AtomicInteger 的值

AtomicInteger 类还提供了一些减小 AtomicInteger 的值的原子性方法。这些方法是：

decrementAndGet()

getAndDecrement()

decrementAndGet() 将 AtomicInteger 的值减一，并返回减一后的值。getAndDecrement() 也将 AtomicInteger 的值减一，但它返回的是减一之前的值。

24. 原子性长整型 AtomicLong

AtomicLong 类为我们提供了一个可以进行原子性读和写操作的 long 变量，它还包含一系列先进的原子性操作，比如 compareAndSet()AtomicLong 类位于 java.util.concurrent.atomic 包，因此其完整类名为 java.util.concurrent.atomic.AtomicLong。本小节描述的 AtomicLong 是 Java 8 版本里的，而不是它第一次被引入的 Java 5 版本。  
AtomicLong 背后的设计理念在我的《Java 并发指南》主题的《[比较和交换](http://tutorials.jenkov.com/java-concurrency/compare-and-swap.html)》小节有解释。

创建一个 AtomicLong

创建一个 AtomicLong 如下：

AtomicLong atomicLong = new AtomicLong();

将创建一个初始值为 0 的 AtomicLong。  
如果你想创建一个指定初始值的 AtomicLong，可以：

AtomicLong atomicLong = new AtomicLong(123);

本示例将 123 作为参数传递给 AtomicLong 的构造子，后者将 AtomicLong 实例的初始值设置为 123。

获取 AtomicLong 的值

你可以通过 get() 方法获取 AtomicLong 的值。AtomicLong.get() 示例：

AtomicLong atomicLong = new AtomicLong(123);

long theValue = atomicLong.get();

设置 AtomicLong 的值

你可以通过 set() 方法设置 AtomicLong 实例的值。一个 AtomicLong.set() 的示例：

AtomicLong atomicLong = new AtomicLong(123);

atomicLong.set(234);

本示例新建了一个初始值为 123 的 AtomicLong，第二行将其值设置为 234。

比较并设置 AtomicLong 的值

AtomicLong 类也有一个原子性的 compareAndSet() 方法。这一方法将 AtomicLong 实例的当前值与一个期望值进行比较，如果两种相等，为 AtomicLong 实例设置一个新值。AtomicLong.compareAndSet() 使用示例：

AtomicLong atomicLong = new AtomicLong(123);

long expectedValue = 123;

long newValue = 234;

atomicLong.compareAndSet(expectedValue, newValue);

本示例新建了一个初始值为 123 的 AtomicLong。然后将 AtomicLong 的当前值与期望值 123 进行比较，如果相等的话，AtomicLong 的新值将变为 234。

增加 AtomicLong 值

AtomicLong 具备一些能够增加 AtomicLong 的值并返回自身值的方法。这些方法如下：

addAndGet()

getAndAdd()

getAndIncrement()

incrementAndGet()

第一个方法 addAndGet() 将 AtomicLong 的值加一个数字，并返回增加后的值。第二个方法 getAndAdd() 也将 AtomicLong 的值加一个数字，但返回的是增加前的 AtomicLong 的值。具体使用哪一个取决于你自己的场景。示例如下：

AtomicLong atomicLong = new AtomicLong();

System.out.println(atomicLong.getAndAdd(10));

System.out.println(atomicLong.addAndGet(10));

本示例将打印出 0 和 20。例子中，第二行拿到的是加 10 之前的 AtomicLong 的值。加 10 之前的值是 0。第三行将 AtomicLong 的值再加 10，并返回加操作之后的值。该值现在是为 20。  
你当然也可以使用这俩方法为 AtomicLong  添加负值。结果实际是一个减法操作。  
getAndIncrement() 和 incrementAndGet() 方法类似于 getAndAdd() 和 addAndGet()，但每次只将 AtomicLong 的值加 1。

减小 AtomicLong 的值

AtomicLong 类还提供了一些减小 AtomicLong  的值的原子性方法。这些方法是：

decrementAndGet()

getAndDecrement()

decrementAndGet() 将 AtomicLong  的值减一，并返回减一后的值。getAndDecrement() 也将 AtomicLong  的值减一，但它返回的是减一之前的值。

25. 原子性引用型 AtomicReference

AtomicReference 提供了一个可以被原子性读和写的对象引用变量。原子性的意思是多个想要改变同一个 AtomicReference 的线程不会导致 AtomicReference 处于不一致的状态。AtomicReference 还有一个 compareAndSet() 方法，通过它你可以将当前引用于一个期望值(引用)进行比较，如果相等，在该 AtomicReference 对象内部设置一个新的引用。

创建一个 AtomicReference

创建 AtomicReference 如下：

AtomicReference atomicReference = new AtomicReference();

如果你需要使用一个指定引用创建 AtomicReference，可以：

String initialReference = "the initially referenced string";

AtomicReference atomicReference = new AtomicReference(initialReference);

创建泛型 AtomicReference

你可以使用 Java 泛型来创建一个泛型 AtomicReference。示例：

AtomicReference<String> atomicStringReference =

new AtomicReference<String>();

你也可以为泛型 AtomicReference 设置一个初始值。示例：

String initialReference = "the initially referenced string";

AtomicReference<String> atomicStringReference =

new AtomicReference<String>(initialReference);

获取 AtomicReference 引用

你可以通过 AtomicReference 的 get() 方法来获取保存在 AtomicReference 里的引用。如果你的 AtomicReference 是非泛型的，get() 方法将返回一个 Object 类型的引用。如果是泛型化的，get() 将返回你创建 AtomicReference 时声明的那个类型。  
先来看一个非泛型的 AtomicReference get() 示例：

AtomicReference atomicReference = new AtomicReference("first value referenced");

String reference = (String) atomicReference.get();

注意如何对 get() 方法返回的引用强制转换为 String。  
泛型化的 AtomicReference 示例：

AtomicReference<String> atomicReference =

new AtomicReference<String>("first value referenced");

String reference = atomicReference.get();

编译器知道了引用的类型，所以我们无需再对 get() 返回的引用进行强制转换了。

设置 AtomicReference  引用

你可以使用 get() 方法对 AtomicReference 里边保存的引用进行设置。如果你定义的是一个非泛型 AtomicReference，set() 将会以一个 Object 引用作为参数。如果是泛型化的 AtomicReference，set() 方法将只接受你定义给的类型。  
AtomicReference set() 示例：

AtomicReference atomicReference =

new AtomicReference();

atomicReference.set("New object referenced");

这个看起来非泛型和泛型化的没啥区别。真正的区别在于编译器将对你能够设置给一个泛型化的 AtomicReference 参数类型进行限制。

比较并设置 AtomicReference 引用

AtomicReference 类具备了一个很有用的方法：compareAndSet()。compareAndSet() 可以将保存在 AtomicReference 里的引用于一个期望引用进行比较，如果两个引用是一样的(并非 equals() 的相等，而是 == 的一样)，将会给 AtomicReference 实例设置一个新的引用。  
如果 compareAndSet() 为 AtomicReference 设置了一个新的引用，compareAndSet() 将返回 true。否则 compareAndSet() 返回 false。  
AtomicReference compareAndSet() 示例：

String initialReference = "initial value referenced";

AtomicReference<String> atomicStringReference =

new AtomicReference<String>(initialReference);

String newReference = "new value referenced";

boolean exchanged = atomicStringReference.compareAndSet(initialReference, newReference);

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

exchanged = atomicStringReference.compareAndSet(initialReference, newReference);

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

# [F5与Nginx的区别](http://www.cnblogs.com/sunyuhuan/p/7280085.html)

**什么是Nginx：**

Nginx：高性能的 HTTP和反向代理服务器，同时支持作为IMAP/POP3/SMTP代理服务器。目前被很多网站应用为其HTTP软负载均衡器。高效的性能、良好的稳定性、丰富的功能集、示例配置文件和低系统资源的消耗正逐渐被大型互联网公司所青睐。例如腾讯、淘宝、新浪等大型门户及商业网站都采用Nginx进行HTTP网站的数据分流。

**Nginx功能特点：**

1、工作在网络的7层之上，可以针对http应用做一些分流的策略，比如针对域名、目录结构；2、Nginx对网络的依赖比较小；3、Nginx安装和配置比较简单，测试起来比较方便；

4、可以承担高的负载压力且稳定，一般能支撑超过1万次的并发；

5、Nginx可以通过端口检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等，并且会把返回错误的请求重新提交到另一个节点，不过其中缺点就是不支持url来检测；

6、Nginx对请求的异步处理可以帮助节点服务器减轻负载；7、Nginx能支持http和Email；

**Nginx，软负载**

优点：基于系统与应用的负载均衡，能够更好地根据系统与应用的状况来分配负载。这对于复杂应用是很重要的，性价比高，实际上如果几台服务器，用F5之类的硬件产品显得有些浪费，而用软件就要合算得多，因为服务器同时还可以跑应用、做集群等。

缺点：负载能力受服务器本身性能的影响，性能越好，负载能力越大。

**F5，硬件**

优点：能够直接通过智能交换机实现,处理能力更强，而且与系统无关，负载性能强，更适用于一大堆设备、大访问量、简单应用。

缺点：成本高，除设备价格高昂，而且配置冗余，很难想象后面服务器做一个集群，但最关键的负载均衡设备却是单点配置，无法有效掌握服务器及应用状态。

硬件负载均衡，一般都不管实际系统与应用的状态，而只是从网络层来判断，所以有时候系统处理能力已经不行了，但网络可能还来得及反应（这种情况非常典型，比如应用服务器后面内存已经占用很多，但还没有彻底不行，如果网络传输量不大就未必在网络层能反映出来）

Nginx推荐的配置是，一个工作进程对应一个CPU内核，确保硬件资源的有效利用。

一旦NGINX服务起来，仅有工作进程在忙，每个工作进程采用非阻塞地方式处理多个连接，降低上下文切换的次数。

每个工作进程都是单线程且独立运行，负责获取新连接并进行处理。进程之间通过共享内存进行通信，诸如缓存数据，会话持续化数据(ession persistence data),以及其他共享资源。

**F5的负载均衡功能**

其实看到Nginx的原理和功能，是不是觉得已经不需要F5了？当然也不是，F5毕竟是负载均衡的老前辈，一直以来都以功能强大，性能稳定著称，很多功能其实是软负载无法做到的。

F5 BIG-IP用作HTTP负载均衡器的主要功能：

1、F5 BIG-IP提供12种灵活的算法将所有流量均衡的分配到各个服务器，而面对用户，只是一台虚拟服务器。

2、F5 BIG-IP可以确认应用程序能否对请求返回对应的数据。假如F5 BIG-IP后面的某一台服务器发生服务停止、死机等故障，F5会检查出来并将该服务器标识为宕机，从而不将用户的访问请求传送到该台发生故障的服务器上。这样，只要其它的服务器正常，用户的访问就不会受到影响。宕机一旦修复，F5 BIG-IP就会自动查证应用保证对客户的请求作出正确响应并恢复向该服务器传送。

3、F5 BIG-IP具有动态Session的会话保持功能，笔者也是在网站中使用的F5将用户IP与Session通过F5进行的绑定，使其Session保持一致。

4、F5 BIG-IP的iRules功能可以做HTTP内容过滤，根据不同的域名、URL，将访问请求传送到不同的服务器。

**各自的原理**

Nginx采用的是反向代理技术，代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个服务器。反向代理负载均衡技术是把将来自internet上的连接请求以反向代理的方式动态地转发给内部网络上的多台服务器进行处理，从而达到负载均衡的目的。

具体是怎么运行的呢？其实当Nginx启动后，其工作进程是由配置文件对其进行初始化的，主进程处理配置文件中的读取、端口绑定等特权操作，之后创建一小组子进程，由这些子进程进行请求的处理，同时缓存加载器加载硬盘中缓存到内存中，接着退出，保证资源开销始终保持着较低的状态。

可以看出，创建的子进程其实在负责所有的工作，处理网络连接、硬盘读写操作、以及上游服务器通信。