# 基础篇

## Java基础

### 面向对象的特征

继承、封装和多态

### final, finally, finalize 的区别

#### final修饰符（关键字）

##### Final修饰类

如果一个类被final修饰，意味着该类不能派生出新的子类，不能作为父类被继承。因此一个类不能被声明为abstract，又被声明为final。

##### Final修饰方法

被声明为final的方法也只能使用，不能重写。

##### Final修饰变量

final修饰一个成员变量（属性），必须要显示初始化。这里有两种初始化方式，一种是在变量声明的时候初始化；第二种方法是在声明变量的时候不赋初值，但是要在这个变量所在的类的所有的构造函数中对这个变量赋初值。

#### Finally

finally作为异常处理的一部分，它只能用在try/catch语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下。

#### Finilize

finalize()是在java.lang.Object里定义的，也就是说每一个对象都有这么个方法。这个方法在gc启动，该对象被回收的时候被调用。其实gc可以回收大部分的对象，所以一般是不需要程序员去实现finalize的。

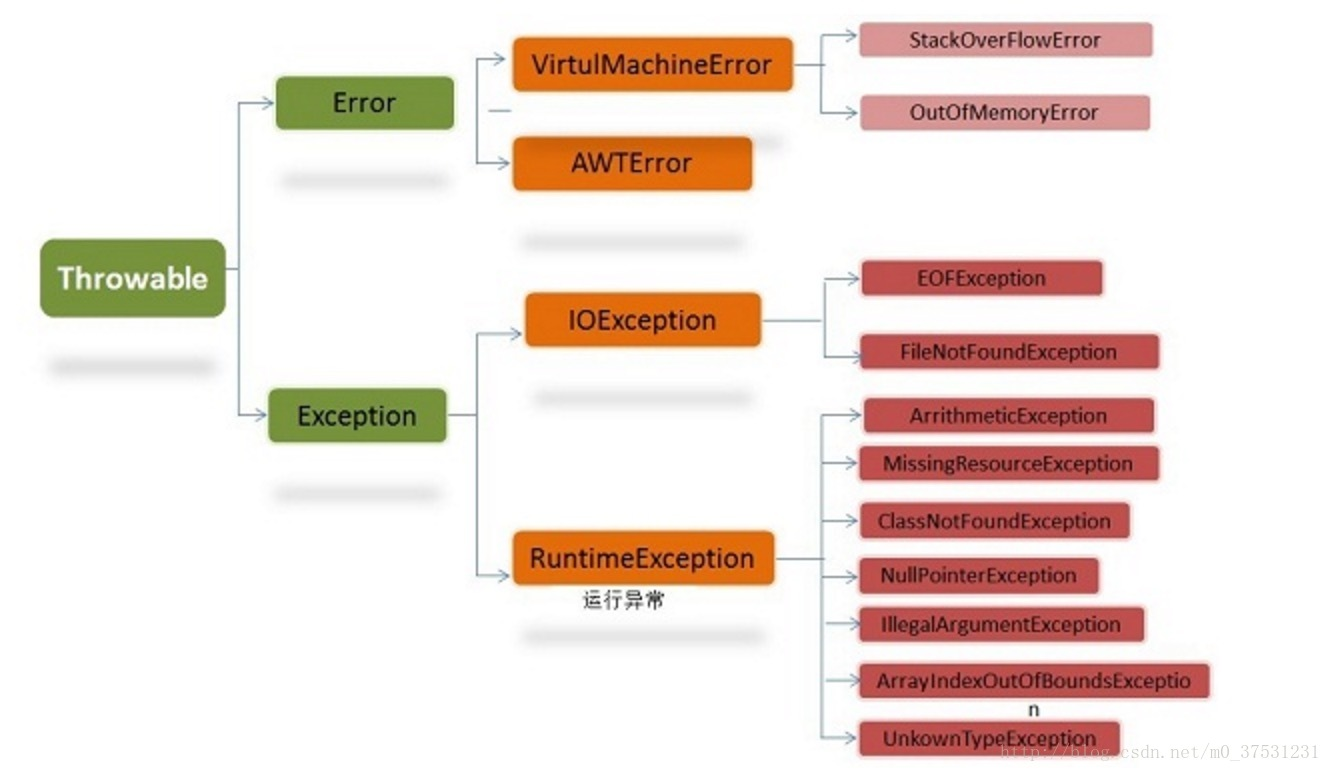
特殊情况下，需要程序员实现finalize，当对象被回收的时候释放一些资源，比如：一个socket链接，在对象初始化时创建，整个生命周期内有效，那么就需要实现finalize，关闭这个链接。

使用finalize还需要注意一个事，调用super.finalize();

一个对象的finalize()方法只会被调用一次，而且finalize()被调用不意味着gc会立即回收该对象，所以有可能调用finalize()后，该对象又不需要被回收了，然后到了真正要被回收的时候，因为前面调用过一次，所以不会调用finalize()，产生问题。 所以，推荐不要使用finalize()方法，它跟析构函数不一样。

### Exception、Error、运行时异常与一般异常有何异同

#### 图



#### Error

描述了java运行时系统的内部错误和资源耗尽错误。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。应用程序不应该抛出这种类型的对象。

#### Exception

##### RuntimeException

##### 非RuntimeException

### 请写出5种常见到的runtime exception

#### ClassCastException

Object x = new Integer(0);

System.out.println((String)x);

#### ArithmeticException

int a=5/0;

#### NullPointerException

String s=null;

int size=s.size();

#### ArrayIndexOutOfBoundsException

#### StringIndexOutOfBoundsException

"hello".indexOf(-1);

#### NegativeArraySizeException

String[] ss=new String[-1];

#### UnsupportedOperationException

#### ConcurrentModificationException

#### FileSystemNotFoundException

#### DOMException

#### ParseException

#### WebServiceException

### 请列出５种常见的Error

#### OutOfMemoryError

#### StackOverflowError

#### NoSuchMethodError

#### NoSuchFieldError

#### UnsupportedClassVersionError

### 请写出５种常见的Checked异常

#### DataFormatException

#### FileNotFoundException

#### EOFException

#### XMLParseException

#### ClassNotFoundException

#### NoSuchFieldException

属性不存在异常。当访问某个类的不存在的属性时抛出该异常。

#### NoSuchMethodException

#### UnsupportedEncodingException

### int 和 Integer 有什么区别，Integer的值缓存范围

Integer是int的包装类；int是基本数据类型；

Integer变量必须实例化后才能使用；int变量不需要；

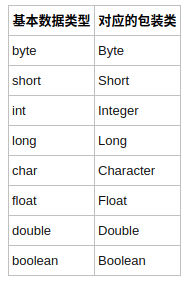
Integer实际是对象的引用，指向此new的Integer对象；int是直接存储数据值 ；

Integer的默认值是null；int的默认值是0。

java在编译Integer i = 100 ;时，会翻译成为Integer i = Integer.valueOf(100)。而java API中对Integer类型的valueOf的定义如下，对于-128到127之间的数，会进行缓存

### 包装类，装箱和拆箱

#### 基本数据类型及对应的包装类



#### 装箱和拆箱

Integer i = 10; //装箱

int n = i; //拆箱

在装箱的时候自动调用的是Integer的valueOf(int)方法。而在拆箱的时候自动调用的是Integer的intValue方法。

### String、StringBuilder、StringBuffer

#### 执行速度

StringBuilder > StringBuffer > String

#### 线程安全

StringBuilder：线程非安全的

StringBuffer：线程安全的

### 重载和重写的区别

#### 重写(Override)

重写是子类对父类的允许访问的方法的实现过程进行重新编写。

访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。

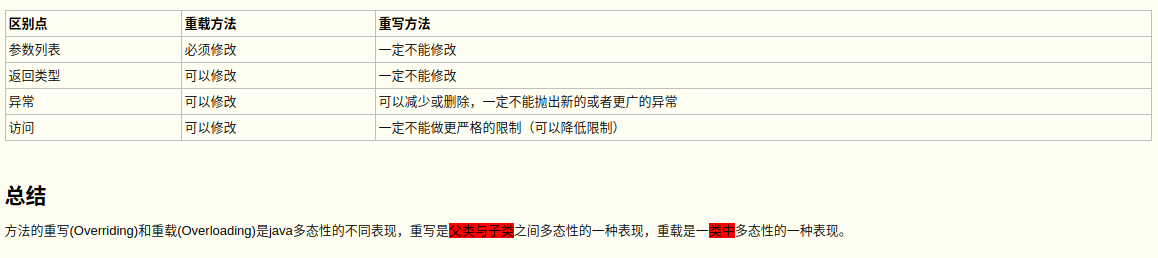
重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如： 父类的一个方法申明了一个检查异常 IOException，但是在重写这个方法的时候不能抛出 Exception 异常，因为 Exception 是 IOException 的父类，只能抛出 IOException 的子类异常。

#### 重载(Overload)

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

法以返回值类型作为重载函数的区分标准。

#### 区别



### 抽象类和接口有什么区别

#### 抽象类

抽象方法是一种特殊的方法：它只有声明，而没有具体的实现。

如果一个类含有抽象方法，则称这个类为抽象类，抽象类必须在类前用abstract关键字修饰。

抽象方法必须为public或者protected（因为如果为private，则不能被子类继承，子类便无法实现该方法），缺省情况下默认为public。

如果一个类继承于一个抽象类，则子类必须实现父类的抽象方法。如果子类没有实现父类的抽象方法，则必须将子类也定义为为abstract类。

#### 接口

接口中的变量会被隐式地指定为public static final变量（并且只能是public static final变量，用private修饰会报编译错误）。

方法会被隐式地指定为public abstract方法且只能是public abstract方法（用其他关键字，比如private、protected、static、 final等修饰会报编译错误）。

#### 抽象类和接口的对比

语法层面上的区别

　　1）抽象类可以提供成员方法的实现细节，而接口中只能存在public abstract 方法；

　　2）抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是public static final类型的；

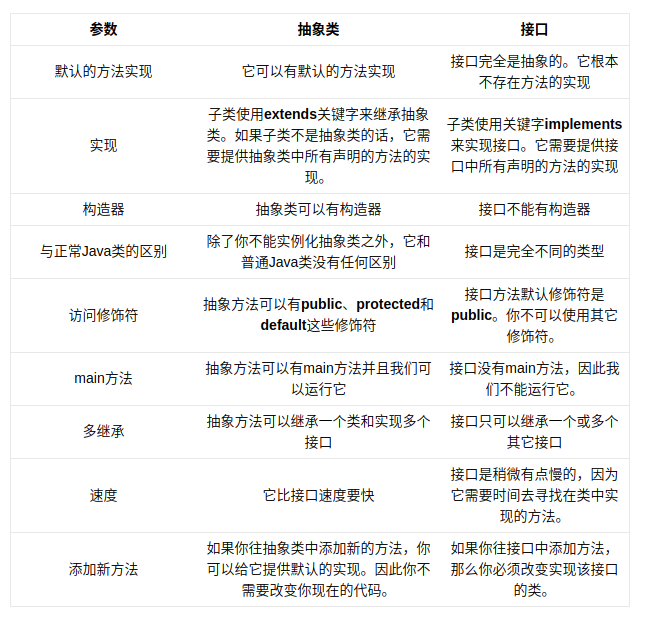
　　3）接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法；

　　4）一个类只能继承一个抽象类，而一个类却可以实现多个接口。

设计层面上的区别

1）抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象。

2）设计层面不同，抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。



### 说说反射的用途及实现

反射机制提供的功能主要有：获取类字段、获取类方法、获取对应的实例构造器，并生成类实例、通过newInstance()方法生成类实例、设置私有变量、获取私有变量。

#### 获取一个对象对应的反射类，在Java中有三种方法可以获取一个对象的反射类

通过getClass()方法

通过Class.forName()方法；

使用类.class

通过类加载器实现，getClassLoader()

### 说说自定义注解的场景及实现

#### 定义注解格式

public @interface 注解名 {定义体}

#### 元注解（meta-annotation）

@Target(ElementType.TYPE) :接口、类、枚举、注解

@Target(ElementType.FIELD) :字段、枚举的常量

@Target(ElementType.METHOD) :方法

@Target(ElementType.PARAMETER) :方法参数

@Target(ElementType.CONSTRUCTOR) :构造函数

@Target(ElementType.LOCAL\_VARIABLE) :局部变量

@Target(ElementType.ANNOTATION\_TYPE):注解

@Target(ElementType.PACKAGE) :包

@Retention(RetentionPolicy.SOURCE) :在源文件中有效（即源文件保留）

@Retention(RetentionPolicy.CLASS) :在class文件中有效（即class保留）

@Retention(RetentionPolicy.RUNTIME) :在运行时有效（即运行时保留）

@Inherited：说明子类可以继承父类中的该注解

@Document：说明该注解将被包含在javadoc

#### 示例

package com.sinitek.common.db;

import java.lang.annotation.\*;

/\*\*

\* Created by wfwt on 2018/3/14 0014.

\*/

@Retention(RetentionPolicy.RUNTIME)

@Inherited

@Target({ElementType.TYPE, ElementType.METHOD})

public @interface UseDataSource {

String value() default DataSourceConstants.MASTER;

}

### HTTP请求的GET与POST方式的区别

#### GET方法

使用GET方法时，查询字符串（键值对）被附加在URL地址后面一起发送到服务器：/test/demo\_form.jsp?name1=value1&name2=value2

特点：

GET请求能够被缓存

GET请求会保存在浏览器的浏览记录中

以GET请求的URL能够保存为浏览器书签

GET请求有长度限制

·GET请求主要用以获取数据

#### POST方法

使用POST方法时，查询字符串在POST信息中单独存在，和HTTP请求一起发送到服务器：

POST/test/demo\_form.jsp HTTP/1.1

Host:w3schools.com

name1=value1&name2=value2

特点：

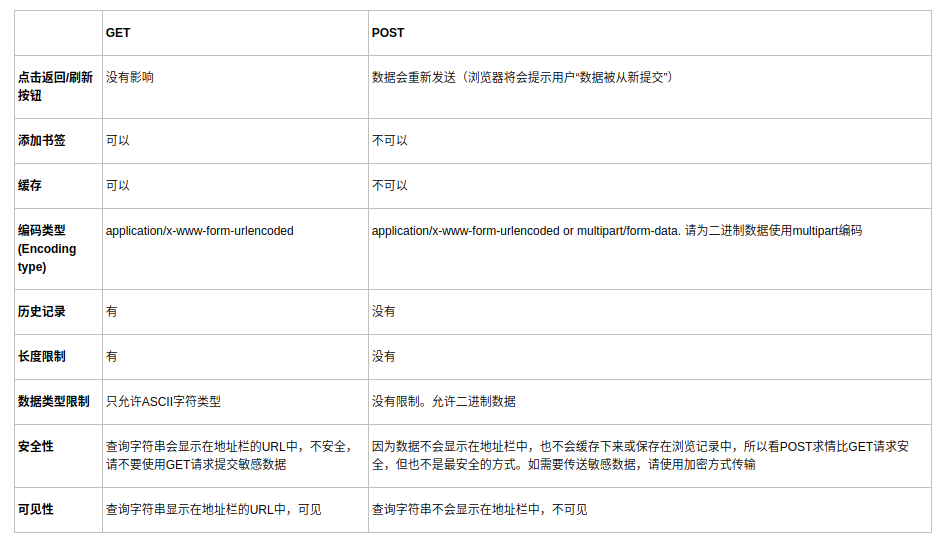
POST请求不能被缓存下来

POST请求不会保存在浏览器浏览记录中

以POST请求的URL无法保存为浏览器书签

POST请求没有长度限制

#### GET和POST的区别



#### Java HTTP get/post

<https://www.cnblogs.com/xjh713/p/6075844.html>

### Session与Cookie区别

#### session的原理

http是无状态的协议，客户每次读取web页面时，服务器都打开新的会话，而且服务器也不会自动维护客户的上下文信息。

session就是一种保存上下文信息的机制，她是针对每一个用户的，session的内容在服务器端，通过sessionId来区分不同的客户，session是以cookie或url重写为基础的，默认用cookie来实现，系统会创造一个JSESSIONID的输出cookie，我们成为session cookie，以区分persistent coookies，注意session cookie是存储于浏览器内存中的，并不是写到硬盘上的；我们通常是看不见JSESSIONID的，但是当我们禁用浏览器的cookie后，web服务器会采用url重写的方式传递sessionid，我们就可以在浏览器看到sessionid=HJHADKSFHKAJSHFJ之类的字符串；session cookie针对某一次会话而言，会话结束session cookie也就消失了。

#### session与cookie的区别

session保存在服务器，客户端不知道其中的信息；cookie保存在客户端，服务端可以知道其中的信息

session中保存的是对象，cookie中保存的是字符串

session不能区分路径，同一个用户在访问一个网站期间，所有的session在任何一个地方都可以访问到；而cookie中如果设置了路径参数，那么同一个网站中不同路径下的cookie互相是访问不到的。

### 列出自己常用的JDK包

### MVC设计思想

model view controller

1)模型: 业务逻辑包含了业务数据的加工与处理以及相应的基础服务(为了保证业务逻辑能够正常进行的事务、安全、权限、日志等等的功能模块)

2)视图:展现模型处理的结果；另外，还要提供相应的操作界面，方便用户使用。

3)控制器:视图发请求给控制器，由控制器来选择相应的模型来处理；模型返回的结果给控制器，由控制器选择合适的视图。

### equals与==的区别

"=="除了比较基本数据之外都是比较的内存地址

"equals"除了没有没有重写equals方法的类之外都是比较的内容（没有重写equals方法的类都是调用的Object的equals的方法，类似＝＝）

### hashCode和equals方法的区别与联系

#### 为什么覆盖(不是重载)equals方法一定要覆盖hashCode方法

因为如果只覆盖了equals而没有覆盖hashCode, 则两个不同的instance a和b虽然equals结果(业务逻辑上)相等,但却会有不同的hashcode,这样hashmap里面会同时存在a和b,而实际上我们需要hashmap里面只能保存其中一个,因为从业务逻辑方向看它们是相等的.

### 什么是Java序列化和反序列化，如何实现Java序列化？或者请解释Serializable 接口的作用

#### Java 序列化和反序列化

把JAVA对象转换为字节序列的过程就称为对象的序列化，将字节序列恢复成Java对象的过程称为对象的反序列化。

ObjectOutputStream类：它继承OutputStream类并且实现ObjectOutput接口。利用该类来实现将对象存储（调用ObjectOutput接口中的writeObject()方法）。ObjectInputStream类：它继承InputStream类并且实现ObjectInput接口。利用该类来实现读取一个对象（调用ObjectInput接口中的readObject()方法）。

#### Serializable和Externalizable

只有实现了 serializable和Externalizable接口的类的对象才能被序列化 后者是前者的子类 实现这个借口的类完全由自身来控制序列化的行为，而仅仅实现前者的类可以采用默认的序列化方式。

如果某个类能够被串行化，其子类也可以被串行化。如果该类有父类，则分两种情况来考虑，如果该父类已经实现了可串行化接口。则其父类的相应字段及属性的处理和该类相同；如果该类的父类没有实现可串行化接口，则该类的父类所有的字段属性将不会串行化。

声明为static和transient类型的成员数据不能被串行化。因为static代表类的状态， transient代表对象的临时数据；

### Object类中常见的方法，为什么wait、notify会放在Object里边?

public final native Class<?> getClass();

public native int hashCode();

public boolean equals(Object obj) { return (this == obj); }

protected native Object clone() throws CloneNotSupportedException;

public String toString() {return getClass().getName() + "@" + Integer.toHexString(hashCode());}

public final native void notify();

public final native void notifyAll();

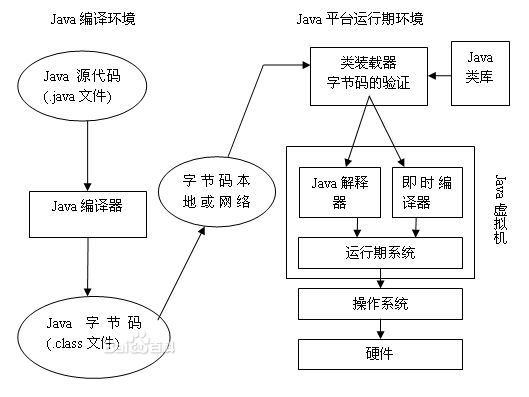
public final native void wait(long timeout) throws InterruptedException;

public final void wait(long timeout, int nanos) throws InterruptedException;

public final void wait() throws InterruptedException { wait(0);}

protected void finalize() throws Throwable { }

### Java的平台无关性如何体现出来的



### JDK和JRE的区别

JDK是Java Development Kit的缩写，是Java的开发工具包。

JRE是Java Runtime Environment的缩写，是Java程序的运行环境。

JDK和JRE是Java开发和运行工具，其中JDK包含了JRE，但是JRE是可以独立安装的。

JDK是Java的开发工具，它不仅提供了Java程序运行所需的JRE，还提供了一系列的编译，运行等工具，如javac，java，javaw等。JRE只是Java程序的运行环境，它最核心的内容就是JVM（Java虚拟机）及核心类库。

### Java 8有哪些新特性

<http://www.importnew.com/11908.html>

#### Lambda表达式与Functional接口

一个lambda可以由用逗号分隔的参数列表、–>符号与函数体三部分表示。

e -> System.out.println( e )

( String e ) -> System.out.println( e )

Lambda可以引用类的成员变量与局部变量（如果这些变量不是final的话，它们会被隐含的转为final，这样效率更高）

函数式接口就是一个具有一个方法的普通接口。像这样的接口，可以被隐式转换为lambda表达式。

默认方法与静态方法并不影响函数式接口的契约，可以任意使用。

#### 接口的默认方法与静态方法

Public interface Defaulable {

default String notRequired() {

return "Default implementation";

}

static Defaulable create( Supplier< Defaulable > supplier ) {

return supplier.get();

}

}

#### 方法引用

##### 构造器引用

它的语法是Class::new，或者更一般的Class< T >::new。请注意构造器没有参数。

##### 静态方法引用

它的语法是Class::static\_method。请注意这个方法接受一个Car类型的参数。

##### 特定类的任意对象的方法引用

它的语法是Class::method。请注意，这个方法没有参数。

##### 特定对象的方法引用

它的语法是instance::method。请注意，这个方法接受一个Car类型的参数

#### 重复注解

@Target( ElementType.TYPE )

    @Retention( RetentionPolicy.RUNTIME )

    public @interface Filters {

        Filter[] value();

    }

    @Target( ElementType.TYPE )

    @Retention( RetentionPolicy.RUNTIME )

    @Repeatable( Filters.class )

    public @interface Filter {

        String value();

    };

    @Filter( "filter1" )

    @Filter( "filter2" )

    public interface Filterable {

}

反射相关的API提供了新的函数getAnnotationsByType()来返回重复注解的类型（请注意Filterable.class.getAnnotation( Filters.class )经编译器处理后将会返回Filters的实例）。

#### 更好的类型推测机制

#### 扩展注解的支持

ElementType.TYPE\_USE和ElementType.TYPE\_PARAMETER是两个新添加的用于描述适当的注解上下文的元素类型。

#### Java编译器的新特性

##### 参数名字

–parameters参数来编译类

#### Java 类库的新特性

##### Optional

##### Stream

##### Date/Time API (JSR 310)

##### JavaScript引擎Nashorn

##### Base64

##### 并行（parallel）数组

#### 新的Java工具

##### Nashorn引擎: jjs

##### 类依赖分析器jdeps

## Java常见集合

### List 和 Set 区别

Iterable<E> <--- Collection<E> <--- List<E>

Iterable<E> <--- Collection<E> <--- Set<E>

### Set和hashCode以及equals方法的联系

当equals()方法被override时，hashCode()也要被override。按照一般hashCode()方法的实现来说，相等的对象，它们的hash code一定相等。

相等（相同）的对象必须具有相等的哈希码（或者散列码）。

如果两个对象的hashCode相同，它们并不一定相同。

### List 和 Map 区别

interface Map<K,V>

### Arraylist 与 LinkedList 区别

Object <--- AbstractCollection<E> <--- AbstractList<E> <--- ArrayList<E>

AbstractList<E> <--- AbstractSequentialList<E> <--- LinkedList<E>

### ArrayList 与 Vector 区别

Object <--- AbstractCollection<E> <--- AbstractList<E> <--- ArrayList<E>

Object <--- AbstractCollection<E> <--- AbstractList<E> <--- Vector<E>

Vector是线程（Thread）同步（Synchronized）的，所以它也是线程安全的，而Arraylist是线程异步（ASynchronized）的，是不安全的。

ArrayList默认大小是10，Vector默认大小是10。

ArrayList调整大小：int newCapacity = oldCapacity + (oldCapacity >> 1);

Vector调整大小：int newCapacity = oldCapacity + ((capacityIncrement > 0) ? capacityIncrement : oldCapacity);

### HashMap 和 Hashtable 的区别

AbstractMap<K,V> <--- HashMap<K,V>

Dictionary<K,V> <--- Hashtable<K,V>

HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。

HashMap是非synchronized，而Hashtable是synchronized，这意味着Hashtable是线程安全的。

HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。（如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。）

HashMap把Hashtable的contains方法去掉了，改成containsvalue和containsKey。因为contains方法容易让人引起误解。

Hashtable中数组默认大小是11，增加的方式是int newCapacity = (oldCapacity << 1) + 1;。HashMap中hash数组的默认大小是16，而且一定是2的指数。

### HashSet 和 HashMap 区别

AbstractMap<K,V> <--- HashMap<K,V>

AbstractCollection<E><--- AbstractSet<E> <--- HashSet<E>

HashSet底层采用的是HashMap进行实现的

### HashMap 和 ConcurrentHashMap 的区别

ConcurrentHashMap将数据分为多个segment，默认16个（concurrency level），然后每次操作对一个segment加锁，避免多线程锁的几率，提高并发效率。

从ConcurrentHashMap代码中可以看出，它引入了一个“分段锁”的概念，具体可以理解为把一个大的Map拆分成N个小的HashTable，根据key.hashCode()来决定把key放到哪个HashTable中。

在ConcurrentHashMap中，就是把Map分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中。

ConcurrentHashMap不允许Null键。

### HashMap 的工作原理及代码实现，什么时候用到红黑树

初始大小为16，装填因子为0.75，则扩容阈值为12。

当链表大小>=8时，则变成红黑树；如果数组长度<64，则直接扩容2倍数组，不变红黑树。

### 多线程情况下HashMap死循环的问题

<https://coolshell.cn/articles/9606.html>

Java 1.8以后应该不会死循环。

### HashMap出现Hash DOS攻击的问题

<https://coolshell.cn/articles/6424.html>

HashMap变成单链表。

### ConcurrentHashMap 的工作原理及代码实现，如何统计所有的元素个数

### 手写简单的HashMap

### 看过那些Java集合类的源码

## 进程和线程

### 线程和进程的概念、并行和并发的概念

进程是一个程序的实例。每个进程都有自己的虚拟地址空间和控制线程，线程是操作系统调度器(Schduler)分配处理器时间的基础单元。

并发：在单核 CPU 系统中，系统调度在某一时刻只能让一个进程运行，虽然这种调度机制有多种形式（大多数是时间片轮巡为主），但无论如何，要通过不断切换需要运行的进程让其运行的方式叫并发

并行：在多核 CPU 系统中，可以让两个以上的进程同时运行在不同的物理核心上，这种运行的方式就是并行

### 创建线程的方式及实现

#### 继承Thread类创建线程

public class MyThread extends Thread{//继承Thread类

　　public void run(){

　　//重写run方法

　　}

}

new MyThread().start();//创建并启动线程

#### 实现Runnable接口创建线程

public class MyThread2 implements Runnable {//实现Runnable接口

　　public void run(){

　　//重写run方法

　　}

}

new Thread(new MyThread2()).start();

#### 使用Callable和Future创建线程

//使用Lambda表达式创建Callable对象

//使用FutureTask类来包装Callable对象

FutureTask<Integer> future=new FutureTask<Integer>(

　　　　(Callable<Integer>)()->{

　　　　　　return 5;

　　　　}

);

new Thread(task,"有返回值的线程").start();//实质上还是以Callable对象来创建并启动线程

System.out.println("子线程的返回值："+future.get());//get()方法会阻塞，直到子线程执行结束才返回

### 进程间通信的方式

统的进程间通信的方式有大致如下几种：

(1) 管道(PIPE) 管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。

(2) 命名管道(FIFO) 命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令mkfifo或系统调用mkfifo来创建。

(3) 信号量(Semphore)

(4) 消息队列(MessageQueue)

(5) 共享内存(SharedMemory)

(6) Socket

Java如何支持进程间通信。我们把Java进程理解为JVM进程。很明显，传统的这些大部分技术是无法被我们的应用程序利用了（这些进程间通信都是靠系统调用来实现的）。但是Java也有很多方法可以进行进程间通信的。

除了上面提到的Socket之外，当然首选的IPC可以使用Rmi，或者Corba也可以。另外Java nio的MappedByteBuffer也可以通过内存映射文件来实现进程间通信(共享内存)。

### 说说 CountDownLatch、CyclicBarrier 原理和区别

<https://www.cnblogs.com/dolphin0520/p/3920397.html>

<https://blog.csdn.net/tolcf/article/details/50925145>



### 说说 Semaphore 原理

AQS

### 说说 Exchanger 原理

### ThreadLocal 原理分析，ThreadLocal为什么会出现OOM，出现的深层次原理

ThreadLocalMap是一个HashMap，初始大小是16，装填因子是2/3，采用开放定址法解决冲突，如下：

private static int nextIndex(int i, int len) {

return ((i + 1 < len) ? i + 1 : 0);

}

在创建很多线程的线程池情况下，由于ThreadLocalMap的生命周期跟Thread一样长，如果没有手动删除对应key就会导致内存泄漏，而不是因为弱引用。

### 讲讲线程池的实现原理

#### 四种策略

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

#### Executors

Executors.newCachedThreadPool(); //创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE

Executors.newSingleThreadExecutor(); //创建容量为1的缓冲池

Executors.newFixedThreadPool(int); //创建固定容量大小的缓冲池

### 线程池的几种实现方式

#### ThreadPoolExecutor

#### ExecutorService、Executors

可缓存线程池

Executors.newCachedThreadPool();

定长线程池

Executors.newFixedThreadPool(3);

定长线程池，定时器

Executors.newScheduledThreadPool(3);

单线程化的线程池

Executors.newSingleThreadExecutor();

#### ScheduledExecutorService

#### ForkJoinPool

RecursiveAction，用于没有返回结果的任务

RecursiveTask，用于有返回值的任务

### 线程的生命周期，状态是如何转移的

## 锁机制

### 说说线程安全问题，什么是线程安全，如何保证线程安全

### 重入锁的概念，重入锁为什么可以防止死锁

<https://blog.csdn.net/noodleprince/article/details/79826050>

已经锁定的情况下去调用另外一个需要锁定的方法。

### 产生死锁的四个条件（互斥、请求与保持、不剥夺、循环等待）

死锁是指多个进程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。

产生死锁的原因主要是：

（1） 因为系统资源不足。

（2） 进程运行推进的顺序不合适。

（3） 资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

### 如何检查死锁（通过jConsole检查死锁）

<https://blog.csdn.net/u014039577/article/details/52351626>

<https://www.cnblogs.com/ilahsa/archive/2013/06/03/3115410.html>

### volatile 实现原理（禁止指令重排、刷新内存）

<https://www.cnblogs.com/chengxiao/p/6528109.html>

当第二个操作是voaltile写时，无论第一个操作是什么，都不能进行重排序

当第一个操作是volatile读时，不管第二个操作是什么，都不能进行重排序

当第一个操作是volatile写时，第二个操作是volatile读时，不能进行重排序

### synchronized 实现原理（对象监视器）

<https://www.cnblogs.com/paddix/p/5367116.html>

Monitorenter/monitorexit

### synchronized 与 lock 的区别



### AQS同步队列

<https://blog.csdn.net/Thousa_Ho/article/details/78047634>

AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）

AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。

tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。

tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。

tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

### CAS无锁的概念、乐观锁和悲观锁

<https://www.cnblogs.com/qjjazry/p/6581568.html>

悲观锁：

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语synchronized关键字的实现也是悲观锁。

乐观锁：

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

### 常见的原子操作类

<https://www.cnblogs.com/senlinyang/p/7856339.html>

atomic提供了3个类用于原子更新基本类型：分别是AtomicInteger原子更新整形，AtomicLong原子更新长整形，AtomicBoolean原子更新bool值。

atomic里提供了三个类用于原子更新数组里面的元素，分别是：AtomicIntegerArray：原子更新整形数组里的元素、AtomicLongArray：原子更新长整形数组里的元素、AtomicReferenceArray：原子更新引用数组里的元素

### 什么是ABA问题，出现ABA问题JDK是如何解决的

如果在算法中的节点可以被循环使用，那么在使用“比较并交换”指令就可能出现这种问题，在CAS操作中将判断“V的值是否仍然为A？”，并且如果是的话就继续执行更新操作，在某些算法中，如果V的值首先由A变为B，再由B变为A，那么CAS将会操作成功

Java中提供了AtomicStampedReference和AtomicMarkableReference来解决ABA问题

<https://blog.csdn.net/kingson_wu/article/details/47378183>

### 乐观锁的业务场景及实现方式

<https://blog.csdn.net/liujinping617/article/details/50370924>

一般来说如果并发量很高的话，建议使用悲观锁，否则的话就使用乐观锁。

如果并发量很高时使用乐观锁的话，会导致很多的并发事务回滚、操作失败。

总之，冲突几率大用悲观，小就用乐观。

<http://www.hollischuang.com/archives/1537>

CAS乐观锁、版本号。

### Java 8并法包下常见的并发类

<https://www.cnblogs.com/caiguoqing/p/6579611.html>

#### 阻塞队列BlockingQueue

ArrayBlockingQueue：基于数组实现的有界阻塞队列，创建后不能修改队列的大小；

LinkedBlockingQueue：基于链表实现的有界阻塞队列，默认大小为Integer.MAX\_VALUE，有较好的吞吐量，但可预测性差。

PriorityBlockingQueue：具有优先级的无界阻塞队列，不允许插入null，所有元素都必须可比较（即实现Comparable接口）。

SynchronousQueue：只有一个元素的同步队列。若队列中有元素插入操作将被阻塞，直到队列中的元素被其他线程取走。

DelayQueue：无界阻塞队列，每个元素都有一个延迟时间，在延迟时间之后才释放元素。

#### 阻塞双端队列BlockingDueue

LinkedBlockingDeque：基于双向链表实现的有界阻塞队列，默认大小为Integer.MAX\_VALUE，有较好的吞吐量，但可预测性差。

#### 阻塞转移队列TransferQueue

#### 并发容器

#### 线程池

#### 锁

ConcurrentHashMap

CopyOnWriteArrayList

CopyOnWriteArraySet

ArrayBlockingQueue

#### 原子类型

Atomic类，如AtomicInteger、AtomicBoolean

#### 并发工具

### 偏向锁、轻量级锁、重量级锁、自旋锁的概念

<https://blog.csdn.net/zqz_zqz/article/details/70233767/>

<https://www.cnblogs.com/lzh-blogs/p/7477157.html>

#### Java偏向锁(Biased Locking)是Java6引入的一项多线程优化。

偏向锁，顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，这种情况下，就会给线程加一个偏向锁。

如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

#### 轻量级锁

是由偏向所升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，偏向锁就会升级为轻量级锁；

#### 重量级锁Synchronized

#### 自旋锁

自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。

但是线程自旋是需要消耗cup的，说白了就是让cup在做无用功，如果一直获取不到锁，那线程也不能一直占用cup自旋做无用功，所以需要设定一个自旋等待的最大时间。

如果持有锁的线程执行的时间超过自旋等待的最大时间扔没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

## JVM

### JVM运行时内存区域划分

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

[https://www.cnblogs.co/red-code/p/6650366.html](https://www.cnblogs.com/red-code/p/6650366.html)

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

#### 线程共享

Method Area（Non-Heap）（方法区）

Heap（堆）

#### 线程不共享

VMStack

Native Method Stack

程序计数器（Program Counter Register）

### 内存溢出OOM和堆栈溢出SOE的示例及原因、如何排查与解决

<https://blog.csdn.net/pbuzhidaol/article/details/72871898>

java.lang.OutOfMemoryError:Java heap space

java.lang.OutOfMemoryError:GC overhead limit exceeded

java.lang.OutOfMemoryError:Permgen space

java.lang.OutOfMemoryError:Metaspace

#### OOM如何排查

<https://blog.csdn.net/John8169/article/details/55802651>

<https://blog.csdn.net/qq_16681169/article/details/53296137>

-XX:+HeapDumpOnOutOfMemoryError

### 如何判断对象是否可以回收或存活

<https://blog.csdn.net/u011277123/article/details/53908315>

<https://blog.csdn.net/tyronerenekton/article/details/59114835>

<https://www.cnblogs.com/fangfuhai/p/7197750.html>

通过一系列被称为"GC Roots"的对象作为起始点，从这些节点开始向下搜索，搜索走过的路径称为引用链，当一个对象到GC Roots对象没有任何引用链相连，就认为GC Roots到这个对象是不可达的，判定此对象为不可用对象，可以被回收。

在Java中，可作为GC Roots的对象包括下面几种：

1、虚拟机栈中引用的对象；

2、方法区中类静态属性引用的对象；

3、方法区中常量引用的对象；

4、本地方法栈中Native方法引用的对象。

### 常见的GC回收算法及其含义

<https://blog.csdn.net/gaoqiao1988/article/details/53539906>

<https://www.cnblogs.com/ityouknow/p/5614961.html>

#### Serial收集器

串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。新生代、老年代使用串行回收；新生代复制算法、老年代标记-压缩；垃圾收集的过程中会Stop The World（服务暂停）

参数控制：-XX:+UseSerialGC 串行收集器

#### ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本。新生代并行，老年代串行；新生代复制算法、老年代标记-压缩

参数控制：-XX:+UseParNewGC ParNew收集器

-XX:ParallelGCThreads 限制线程数量

#### Parallel收集器

Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。可以通过参数来打开自适应调节策略，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制GC的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩

参数控制：-XX:+UseParallelGC 使用Parallel收集器+ 老年代串行

#### Parallel Old 收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法。这个收集器是在JDK 1.6中才开始提供

参数控制： -XX:+UseParallelOldGC 使用Parallel收集器+ 老年代并行

#### CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

优点:并发收集、低停顿

缺点：产生大量空间碎片、并发阶段会降低吞吐量

参数控制：-XX:+UseConcMarkSweepGC 使用CMS收集器

-XX:+ UseCMSCompactAtFullCollection Full GC后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长

-XX:+CMSFullGCsBeforeCompaction 设置进行几次Full GC后，进行一次碎片整理

-XX:ParallelCMSThreads 设定CMS的线程数量（一般情况约等于可用CPU数量）

#### G1收集器

### 常见的JVM性能监控和故障处理工具类：jps、jstat、jmap、jinfo、jconsole等

<https://www.cnblogs.com/baihuitestsoftware/articles/6382733.html>

#### Jps

Java Virtual Machine Process Status Tool

查看所有的jvm进程，包括进程ID，进程启动的路径等等。

我自己也用PS，即：ps -ef | grep java

#### Jstat

jstat利用JVM内建的指令对Java应用程序的资源和性能进行实时的命令行的监控，包括了对进程的classloader，compiler，gc情况；

特别的，一个极强的监视内存的工具，可以用来监视VM内存内的各种堆和非堆的大小及其内存使用量，以及加载类的数量。

#### Jmap

监视进程运行中的jvm物理内存的占用情况，该进程内存内，所有对象的情况，例如产生了哪些对象，对象数量；

#### Jinfo

观察进程运行环境参数，包括Java System属性和JVM命令行参数

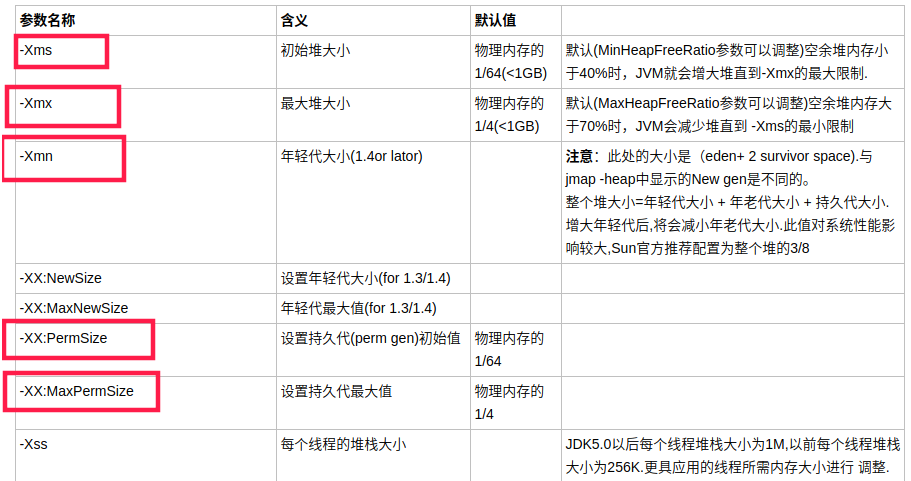
#### Jconsole

jconsole是基于 (JMX)的实时图形化监测工具，这个工具利用了内建到JVM里面的JMX指令来提供实时的性能和资源的监控，包括了Java程序的内存使用，Heap size, 线程的状态，类的分配状态和空间使用等等。

与jstat相当。

### JVM如何设置参数

<https://www.cnblogs.com/jianyungsun/p/6911380.html>



### JVM性能调优

<https://blog.csdn.net/chen77716/article/details/5695893>

内存、垃圾回收算法

### 类加载器、双亲委派模型、一个类的生命周期、类是如何加载到JVM中的

<https://www.cnblogs.com/ityouknow/p/5603287.html>

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

其中类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。

类加载有三种方式：

1、命令行启动应用时候由JVM初始化加载

2、通过Class.forName()方法动态加载

3、通过ClassLoader.loadClass()方法动态加载

### 类加载的过程：加载、验证、准备、解析、初始化

### 强引用、软引用、弱引用、虚引用

<https://www.cnblogs.com/yw-ah/p/5830458.html>

#### 强引用：

只要引用存在，垃圾回收器永远不会回收

Object obj = new Object();

//可直接通过obj取得对应的对象 如obj.equels(new Object());

而这样 obj对象对后面new Object的一个强引用，只有当obj这个引用被释放之后，对象才会被释放掉，这也是我们经常所用到的编码形式。

#### 软引用：

非必须引用，内存溢出之前进行回收，可以通过以下代码实现

Object obj = new Object();

SoftReference<Object> sf = new SoftReference<Object>(obj);

obj = null;

sf.get();//有时候会返回null

这时候sf是对obj的一个软引用，通过sf.get()方法可以取到这个对象，当然，当这个对象被标记为需要回收的对象时，则返回null；

软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

#### 弱引用：

第二次垃圾回收时回收，可以通过如下代码实现

Object obj = new Object();

WeakReference<Object> wf = new WeakReference<Object>(obj);

obj = null;

wf.get();//有时候会返回null

wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾

弱引用是在第二次垃圾回收时回收，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回null。

弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的isEnQueued方法返回对象是否被垃圾回收器标记。

#### 虚引用：

垃圾回收时回收，无法通过引用取到对象值，可以通过如下代码实现

Object obj = new Object();

PhantomReference<Object> pf = new PhantomReference<Object>(obj);

obj=null;

pf.get();//永远返回null

pf.isEnQueued();//返回是否从内存中已经删除

虚引用是每次垃圾回收的时候都会被回收，通过虚引用的get方法永远获取到的数据为null，因此也被成为幽灵引用。

虚引用主要用于检测对象是否已经从内存中删除。

### Java内存模型JMM

<https://blog.csdn.net/suifeng3051/article/details/52611310>

<https://blog.csdn.net/javazejian/article/details/72772461>

Java线程之间的通信采用的是过共享内存模型，这里提到的共享内存模型指的就是Java内存模型(简称JMM)，JMM决定一个线程对共享变量的写入何时对另一个线程可见。

happens-before

从jdk5开始，java使用新的JSR-133内存模型，基于happens-before的概念来阐述操作之间的内存可见性。

在JMM中，如果一个操作的执行结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系，这个的两个操作既可以在同一个线程，也可以在不同的两个线程中。

与程序员密切相关的happens-before规则如下：

程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意的后续操作。

监视器锁规则：对一个锁的解锁操作，happens-before于随后对这个锁的加锁操作。

volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。

传递性规则：如果 A happens-before B，且 B happens-before C，那么A happens-before C。

注意：两个操作之间具有happens-before关系，并不意味前一个操作必须要在后一个操作之前执行！仅仅要求前一个操作的执行结果，对于后一个操作是可见的，且前一个操作按顺序排在后一个操作之前。

## 设计模式

### 常见的设计模式

<http://www.runoob.com/design-pattern/design-pattern-tutorial.html>

设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。

### 设计模式的的六大原则及其含义

设计模式的六大原则：

总原则－开闭原则

对扩展开放，对修改封闭。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。

想要达到这样的效果，我们需要使用接口和抽象类等，后面的具体设计中我们会提到这点。

1、单一职责原则

不要存在多于一个导致类变更的原因，也就是说每个类应该实现单一的职责，否则就应该把类拆分。

2、里氏替换原则（Liskov Substitution Principle）

任何基类可以出现的地方，子类一定可以出现。里氏替换原则是继承复用的基石，只有当衍生类可以替换基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。

里氏代换原则是对“开-闭”原则的补充。实现“开闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。里氏替换原则中，子类对父类的方法尽量不要重写和重载。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

3、依赖倒转原则（Dependence Inversion Principle）

面向接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

4、接口隔离原则（Interface Segregation Principle）

每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

5、迪米特法则（最少知道原则）（Demeter Principle）

一个类对自己依赖的类知道的越少越好。无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

6、合成复用原则（Composite Reuse Principle）

尽量首先使用合成/聚合的方式，而不是使用继承。

### 常见的单例模式以及各种实现方式的优缺点，哪一种最好，手写常见的单利模式

<http://www.runoob.com/design-pattern/singleton-pattern.html>

静态变量

静态变量+synchronized

私有静态类

额外的synchronized方法

### 设计模式在实际场景中的应用

### Spring中用到了哪些设计模式

代理：JdkDynamicAopProxy、CglibAopProxy

观察者：ApplicationListener

工厂模式：BeanFactory

### MyBatis中用到了哪些设计模式

工厂方法：SqlSessionFactory

建造者模式：SqlSessionFactoryBuilder

代理：MapperProxy、MapperProxyFactory

装饰者：Cache、LruCache

### 你项目中有使用哪些设计模式

### 说说常用开源框架中设计模式使用分析

### 动态代理很重要！！！

<https://blog.csdn.net/heyutao007/article/details/49738887>

InvocationHandler

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable;

Proxy.newProxyInstance(calculatorImpl.getClass().getClassLoader(), calculatorImpl.getClass().getInterfaces(), calculatorHandler);

net.sf.cglib.proxy.Enhancer – 主要的增强类

net.sf.cglib.proxy.MethodInterceptor – 主要的方法拦截类，它是Callback接口的子接口，需要用户实现

## 数据结构

### 树（二叉查找树、平衡二叉树、红黑树、B树、B+树）

### 深度有限算法、广度优先算法

### 克鲁斯卡尔算法、普林母算法、迪克拉斯算法

### 什么是一致性Hash及其原理、Hash环问题

<http://www.zsythink.net/archives/1182>

<https://www.jianshu.com/p/e8fb89bb3a61>

求余算法: hash(object)%N

### 常见的排序算法和查找算法：快排、折半查找、堆排序等

<https://blog.csdn.net/zhangxiaoyu_sy/article/details/75000151>

<https://blog.csdn.net/daguairen/article/details/52611874>

排序的稳定性和复杂度

不稳定：

选择排序（selection sort）— O(n2)

快速排序（quicksort）— O(nlogn) 平均时间, O(n2) 最坏情况; 对于大的、乱序串列一般认为是最快的已知排序

堆排序 （heapsort）— O(nlogn)

希尔排序 （shell sort）— O(nlogn)

基数排序（radix sort）— O(n·k); 需要 O(n) 额外存储空间 （K为特征个数）

稳定：

插入排序（insertion sort）— O(n2)

冒泡排序（bubble sort） — O(n2)

归并排序 （merge sort）— O(n log n); 需要 O(n) 额外存储空间

二叉树排序（Binary tree sort） — O(nlogn); 需要 O(n) 额外存储空间

计数排序 (counting sort) — O(n+k); 需要 O(n+k) 额外存储空间，k为序列中Max-Min+1

桶排序 （bucket sort）— O(n); 需要 O(k) 额外存储空间

## 网络/IO基础

### BIO、NIO、AIO的概念

BIO（Blocking IO）阻塞IO

NIO(reactor模型) （Non-Blocking IO）非阻塞IO

AIO(proactor模型)

### 什么是长连接和短连接

### Http1.0和2.0相比有什么区别，可参考《Http 2.0》

<https://blog.csdn.net/u012657197/article/details/77877840>

<https://www.zhihu.com/question/34074946>

HTTP1.0利用文本与服务器交互，而HTTP2的基本协议单位为二进制帧流，每帧都有自己的类型旨在实现不同功能，然而HTTP1将继续存在，因此，HTTP2的二进制帧流需映射到HTTP1上以实现向下兼容。

HTTP2.0这么多新特性就是因为底层数据格式的改变

### Https的基本概念

HTTPS = HTTP + SSL

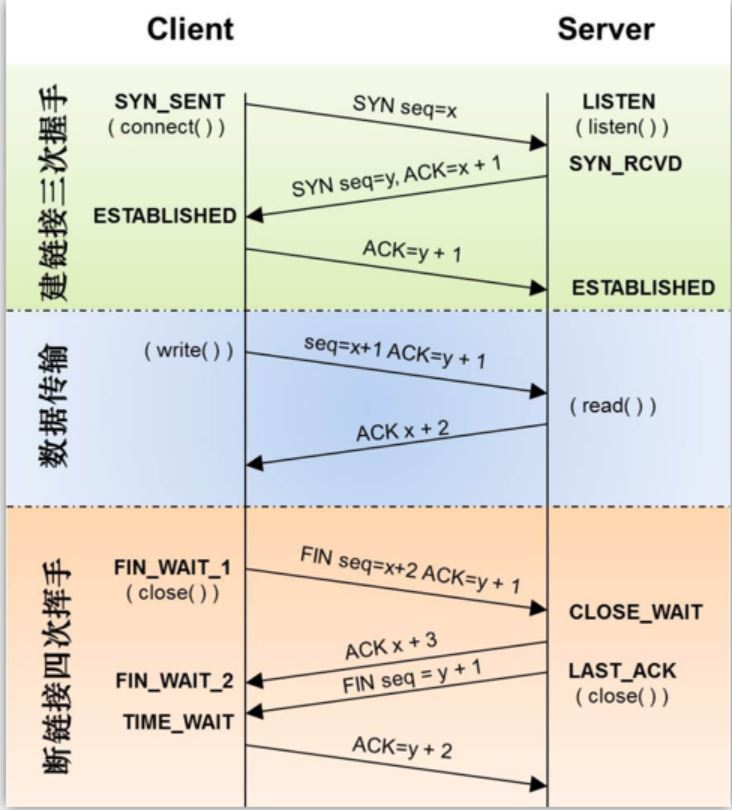
<https://www.zhihu.com/question/25912483>

既然是加密，那肯定是不希望别人知道我的消息，所以只有我才能解密，所以可得出公钥负责加密，私钥负责解密；同理，既然是签名，那肯定是不希望有人冒充我发消息，只有我才能发布这个签名，所以可得出私钥负责签名，公钥负责验证。

### 三次握手和四次挥手、为什么挥手需要四次

<https://www.cnblogs.com/huhuuu/p/3572485.html>

<https://blog.csdn.net/yanxiaolx/article/details/52123628>



MSL（Maximum Segment Lifetime）最大报文生存时间

TTL（time-to-live）生存时间字段

RTT（round-trip-time）往返时间

### 从游览器中输入URL到页面加载的发生了什么？可参考《从输入URL到页面加载发生了什么》

<https://www.cnblogs.com/engeng/articles/5943382.html>

<https://www.cnblogs.com/daijinxue/p/6640153.html>

DNS解析

TCP连接

发送HTTP请求

服务器处理请求并返回HTTP报文

浏览器解析渲染页面

连接结束

### HTTP协议

<https://www.cnblogs.com/qdhxhz/p/8468913.html>

<https://www.cnblogs.com/ranyonsue/p/5984001.html>

# 数据存储和消息队列

## 数据库

### MySQL 索引使用的注意事项

<https://www.cnblogs.com/jianmingyuan/p/6625026.html>

<https://www.cnblogs.com/heyonggang/p/6610526.html>

索引的类型：

UNIQUE(唯一索引)：不可以出现相同的值，可以有NULL值

INDEX(普通索引)：允许出现相同的索引内容

PROMARY KEY(主键索引)：不允许出现相同的值

fulltext index(全文索引)：可以针对值中的某个单词，但效率确实不敢恭维

组合索引：实质上是将多个字段建到一个索引里，列值的组合必须唯一

### DDL、DML、DCL分别指什么

DML（data manipulation language）：

它们是SELECT、UPDATE、INSERT、DELETE，就象它的名字一样，这4条命令是用来对数据库里的数据进行操作的语言

DDL（data definition language）：

DDL比DML要多，主要的命令有CREATE、ALTER、DROP等，DDL主要是用在定义或改变表（TABLE）的结构，数据类型，表之间的链接和约束等初始化工作上，他们大多在建立表时使用

DCL（Data Control Language）：

是数据库控制功能。是用来设置或更改数据库用户或角色权限的语句，包括（grant,deny,revoke等）语句。在默认状态下，只有sysadmin,dbcreator,db\_owner或db\_securityadmin等人员才有权力执行DCL

### explain命令

<https://www.cnblogs.com/gomysql/p/3720123.html>

### left join，right join，inner join

### 数据库事物ACID（原子性、一致性、隔离性、持久性）

<https://www.cnblogs.com/xuanzhi201111/p/4103696.html>

原子性（atomicity）

　　一个事务必须被视为一个不可分割的最小工作单元，整个事务中的所有操作要么全部提交成功，要么全部失败回滚，对于一个事务来说，不可能只执行其中的一部分操作，这就是事务的原子性

一致性（consistency）

数据库总是从一个一致性的状态转换到另一个一致性的状态。（在前面的例子中，一致性确保了，即使在执行第三、四条语句之间时系统崩溃，支票账户中也不会损失200美元，因为事务最终没有提交，所以事务中所做的修改也不会保存到数据库中。）

隔离性（isolation）

通常来说，一个事务所做的修改在最终提交以前，对其他事务是不可见的。（在前面的例子中，当执行完第三条语句、第四条语句还未开始时，此时有另外的一个账户汇总程序开始运行，则其看到支票帐户的余额并没有被减去200美元。）

持久性（durability）

　　一旦事务提交，则其所做的修改不会永久保存到数据库。（此时即使系统崩溃，修改的数据也不会丢失。持久性是个有占模糊的概念，因为实际上持久性也分很多不同的级别。有些持久性策略能够提供非常强的安全保障，而有些则未必，而且不可能有能做到100%的持久性保证的策略。）

### 事物的隔离级别（读未提交、读以提交、可重复读、可序列化读）

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

<https://www.cnblogs.com/huanongying/p/7021555.html>



### 脏读、幻读、不可重复读

### 数据库的几大范式

#### 三大范式

<https://www.cnblogs.com/waj6511988/p/7027127.html>

第一范式（1NF）：数据表中的每一列（每个字段）必须是不可拆分的最小单元，也就是确保每一列的原子性；

第二范式（2NF）：满足1NF后，要求表中的所有列，都必须依赖于主键，而不能有任何一列与主键没有关系，也就是说一个表只描述一件事情；

第三范式（3NF）：必须先满足第二范式（2NF），要求：表中的每一列只与主键直接相关而不是间接相关，（表中的每一列只能依赖于主键）；

#### 如何更好的区分三大范式

第 一范式和第二范式在于有没有分出两张表，第二范式是说一张表中包含了多种不同的实体属性，那么要必须分成多张表， 第三范式是要求已经分成了多张表，那么一张表中只能有另一张表中的id（主键），而不能有其他的任何信息（其他的信息一律用主键在另一表查询）。

#### 数据库五大约束

1.primary KEY:设置主键约束；

2.UNIQUE：设置唯一性约束，不能有重复值；

3.DEFAULT 默认值约束，height DOUBLE(3,2)DEFAULT 1.2 height不输入是默认为1,2

4.NOT NULL：设置非空约束，该字段不能为空；

5.FOREIGN key :设置外键约束。

### 数据库常见的命令

<https://www.2cto.com/database/201610/553534.html>

<https://www.cnblogs.com/zhouzetian/p/6429181.html>

<https://www.cnblogs.com/moss_tan_jun/p/5731901.html>

### 说说分库与分表设计

<https://www.cnblogs.com/firstdream/p/6728106.html>

一个数据库由很多表的构成，每个表对应着不同的业务，垂直切分是指按照业务将表进行分类，分布到不同的数据库上面，这样也就将数据或者说压力分担到不同的库上面

垂直拆分是把不同的表拆到不同的数据库中，而水平拆分是把同一个表拆到不同的数据库中。

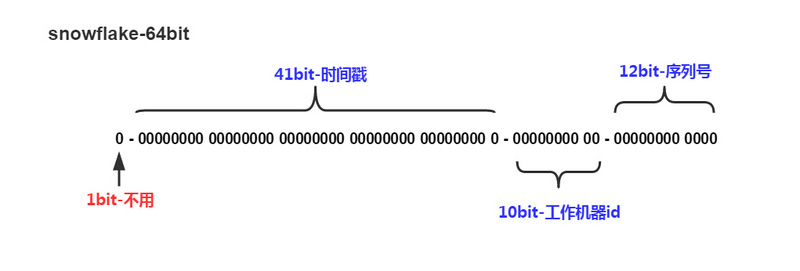
### 分库与分表带来的分布式困境与应对之策（如何解决分布式下的分库分表，全局表？）

#### 表关联问题

#### 分页与排序问题

#### 分布式事务问题

#### 分布式全局唯一ID



### 说说 SQL 优化之道

<http://www.cnblogs.com/yunfeifei/p/3850440.html>

### MySQL遇到的死锁问题、如何排查与解决

<https://blog.csdn.net/hj7jay/article/details/56274056>

<https://www.cnblogs.com/LBSer/p/5183300.html>

### 存储引擎的 InnoDB与MyISAM区别，优缺点，使用场景

<https://blog.csdn.net/liu857279611/article/details/51558927>

<https://blog.csdn.net/aaa123524457/article/details/54375341>

<https://www.cnblogs.com/jeffen/p/5990605.html>

<https://www.cnblogs.com/kevingrace/p/5685355.html>

主要区别：

1).MyISAM是非事务安全型的，而InnoDB是事务安全型的。

2).MyISAM锁的粒度是表级，而InnoDB支持行级锁定。

3).MyISAM支持全文类型索引，而InnoDB不支持全文索引。

4).MyISAM相对简单，所以在效率上要优于InnoDB，小型应用可以考虑使用MyISAM。

5).MyISAM表是保存成文件的形式，在跨平台的数据转移中使用MyISAM存储会省去不少的麻烦。

6).InnoDB表比MyISAM表更安全，可以在保证数据不会丢失的情况下，切换非事务表到事务表（alter table tablename type=innodb）。

### 索引类别（B+树索引、全文索引、哈希索引）、索引的原理

#### 全文索引

<https://blog.csdn.net/u011734144/article/details/52817766>

<https://segmentfault.com/a/1190000006997077>

### 什么是自适应哈希索引（AHI）

InnoDB存储引擎会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升，则建立哈希索引，称之为自适应哈希索引(Adaptive Hash Index, AHI)。AHI是通过缓冲池的B+树页构造而来，因此建立的速度很快，而且不需要对整张表构建哈希索引。InnoDB存储引擎会自动根据访问的频率和模式来自动地为某些热点页建立哈希索引。

### 为什么要用 B+tree作为MySQL索引的数据结构

<https://www.jianshu.com/p/0371c9569736>

### 聚集索引与非聚集索引的区别

<https://www.cnblogs.com/s-b-b/p/8334593.html>

<https://blog.csdn.net/zc474235918/article/details/50580639>

<https://www.cnblogs.com/shanshanlaichi/p/6568097.html>

聚集（clustered）索引，也叫聚簇索引。

定义：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。

非聚集（unclustered）索引。

定义：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。

### 遇到过索引失效的情况没，什么时候可能会出现，如何解决

<http://www.jb51.net/article/50649.htm>

### limit 20000 加载很慢怎么解决

<http://www.jb51.net/article/85312.htm>

<https://blog.csdn.net/qq_36276335/article/details/73824243>

### 如何选择合适的分布式主键方案

<https://blog.csdn.net/chenshun123/article/details/79588208>

<https://www.cnblogs.com/btgyoyo/p/7156589.html>

### 选择合适的数据存储方案

### 常见的几种分布式ID的设计方案

<https://blog.csdn.net/hl_java/article/details/78462283>

<https://www.cnblogs.com/haoxinyue/p/5208136.html>

UUID

多数据库+错开

类snowflake方案

常见的数据库优化方案，在你的项目中数据库如何进行优化的

## Redis

### Redis 有哪些数据类型，可参考《Redis常见的5种不同的数据类型详解》



### Redis 内部结构

<https://www.cnblogs.com/shanheyongmu/p/6282557.html>

Redis内部使用一个redisObject对象来表示所有的key和value。redisObject主要的信息包括数据类型（type）、编码方式(encoding)、数据指针（ptr）、虚拟内存（vm）等。type代表一个value对象具体是何种数据类型，encoding是不同数据类型在redis内部式。

<https://blog.csdn.net/tianshijianbing1989/article/details/50730572>

### Redis 使用场景

<https://www.cnblogs.com/NiceCui/p/7794659.html>

<https://blog.csdn.net/u011277123/article/details/78692603/>

<https://www.cnblogs.com/zhangtianle/p/7403175.html>

缓存、计数器、队列、zset排行榜、最新列表、会话缓存、

### Redis 持久化机制，可参考《使用快照和AOF将Redis数据持久化到硬盘中》

<https://blog.csdn.net/xlgen157387/article/details/61925524>

<https://blog.csdn.net/CSDNones/article/details/50387256>

Redis保存数据有两种方式：

快照模式（Snapshot）

它支持两种快照模式：

定时快照，即按一定时间将内存中的数据保存到磁盘上。

定量快照，即数据变化一定次数后将数据保存到磁盘上。

你也可以结合这两种方式，比如变化1000次，距离上次保存时间60秒以上才保存数据。

写模式（Append Only File）

这种模式下Redis会把所以修改数据的命令（如Update，Set）等保存到一个只能追加的ASAP文件中，当Redis重启时，它会把这个文件里的命令重新执行一遍。

### Redis 集群方案与实现

Redis集群方案显得尤为重要。通常有3个途径：官方Redis Cluster；通过Proxy分片；客户端分片(Smart Client)。

<https://blog.csdn.net/sanpo/article/details/52839044>

<https://blog.csdn.net/KingCat666/article/details/78552511>

<https://blog.csdn.net/u014756827/article/details/52116538>

Redis Sentinel

<https://blog.csdn.net/men_wen/article/details/72724406>

<https://www.cnblogs.com/jeffen/p/6107484.html>

<https://www.cnblogs.com/zhoujinyi/p/5570024.html>

### Redis 为什么是单线程的？

因为CPU不是Redis的瓶颈。Redis的瓶颈最有可能是机器内存或者网络带宽。

### 缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

<https://www.cnblogs.com/leeSmall/p/8594542.html>

<https://blog.csdn.net/qq_26562641/article/details/53004578>

#### 缓存雪崩

我们可以简单的理解为：由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

#### 缓存穿透

是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

#### 缓存预热

就是系统上线后，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

#### 缓存更新

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

（1）定时去清理过期的缓存；

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

#### 缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

### 使用缓存的合理性问题

<https://blog.csdn.net/diyhzp/article/details/54892358>

### Redis常见的回收策略

<https://blog.csdn.net/ligupeng7929/article/details/79603060>

noeviction: 不删除策略, 达到最大内存限制时, 如果需要更多内存, 直接返回错误信息。 大多数写命令都会导致占用更多的内存(有极少数会例外, 如 DEL )。

allkeys-lru: 所有key通用; 优先删除最近最少使用(less recently used ,LRU) 的 key。

volatile-lru: 只限于设置了 expire 的部分; 优先删除最近最少使用(less recently used ,LRU) 的 key。

allkeys-random: 所有key通用; 随机删除一部分 key。

volatile-random: 只限于设置了 expire 的部分; 随机删除一部分 key。

volatile-ttl: 只限于设置了 expire 的部分; 优先删除剩余时间(time to live,TTL) 短的key。

## 消息队列

### 消息队列的使用场景

<https://blog.csdn.net/seven__________7/article/details/70225830>

异步处理

应用解耦

流量削峰

日志处理

消息通讯

### 消息的重发补偿解决思路

<https://blog.csdn.net/zl1zl2zl3/article/details/51966826>

<http://www.dataguru.cn/article-9582-1.html>

版本号

参考TCP/IP，序列号大1的消息

唯一的消息ID

消息乱序（业务层使用状态机）

事务解决方案从大方向上有两种：

两阶段提交，分布式事务。

本地事务，本地落地，补偿发送。

### 消息的幂等性解决思路

只有1条消息被消费，保证了幂等。

### 消息的堆积解决思路

push模型最大问题是慢消费，即消费者速度如果比生产者速度慢很多，会导致消息在broker的堆积，如果这些消息是有用无法丢弃的就会一直在broker端保存，并且broker会不断的给消费者推送消息，消费者reject或error后可能会来回推送。

pull模式最大的问题是消息延迟和忙等。业界较成熟的做法是从短时间开始（不会对broker有太大负担），然后指数级增长等待。比如开始等5ms，然后10ms，然后20ms，然后40ms……直到有消息到来，然后再回到5ms。

处理消息堆积的方法就是把它存下来。只是这个存储可以做成很多方式。比如存储在内存里，存储在分布式KV里，存储在磁盘里，存储在数据库里等等。但归结起来，主要有持久化和非持久化两种。

### 自己如何实现消息队列

<https://www.cnblogs.com/doit8791/p/8570759.html>

<https://github.com/aCoder2013/blog/issues/21?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io>

<https://tech.meituan.com/mq-design.html?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io>

### 如何保证消息的有序性

# 开源框架和容器

## SSM/Servlet

### Servlet的生命周期

<https://blog.csdn.net/danielzhou888/article/details/70835418>

<https://www.cnblogs.com/EasonJim/p/6183856.html>

Servlet 加载—>实例化—>服务—>销毁。

### 转发与重定向的区别

<https://blog.csdn.net/lishehe/article/details/23133105>

转发过程

客户首先发送一个请求到服务器端，服务器端发现匹配的servlet，并指定它去执行，当这个servlet执行完之后，它要调用getRequestDispacther()方法，把请求转发给指定的student\_list.jsp,整个流程都是在服务器端完成的，而且是在同一个请求里面完成的，因此servlet和jsp共享的是同一个request，在servlet里面放的所有东西，在student\_list中都能取出来，因此，student\_list能把结果getAttribute()出来，getAttribute()出来后执行完把结果返回给客户端。整个过程是一个请求，一个响应。

重定向过程

客户发送一个请求到服务器，服务器匹配servlet，这都和请求转发一样，servlet处理完之后调用了sendRedirect()这个方法，这个方法是response的方法，所以，当这个servlet处理完之后，看到response.senRedirect()方法，立即向客户端返回这个响应，响应行告诉客户端你必须要再发送一个请求，去访问student\_list.jsp，紧接着客户端受到这个请求后，立刻发出一个新的请求，去请求student\_list.jsp,这里两个请求互不干扰，相互独立，在前面request里面setAttribute()的任何东西，在后面的request里面都获得不了。可见，在sendRedirect()里面是两个请求，两个响应。

### BeanFactory 和 ApplicationContext 有什么区别

<https://blog.csdn.net/u011202334/article/details/51509235>

<https://www.cnblogs.com/wnlja/p/3907836.html>

Spring提供了两种不同的容器：一种是最基本的BeanFactory，另一种是扩展的ApplicationContext。BeanFactory 仅提供了最基本的依赖注入支持，而 ApplicationContext 则扩展了BeanFactory ,提供了更多的额外功能。

BeanFacotry是spring中比较原始的Factory。如XMLBeanFactory就是一种典型的BeanFactory。原始的BeanFactory无法支持spring的许多插件，如AOP功能、Web应用等。

ApplicationContext接口,它由BeanFactory接口派生而来，因而提供BeanFactory所有的功能。ApplicationContext以一种更向面向框架的方式工作以及对上下文进行分层和实现继承，ApplicationContext包还提供了以下的功能：

• MessageSource, 提供国际化的消息访问

• 资源访问，如URL和文件

• 事件传播

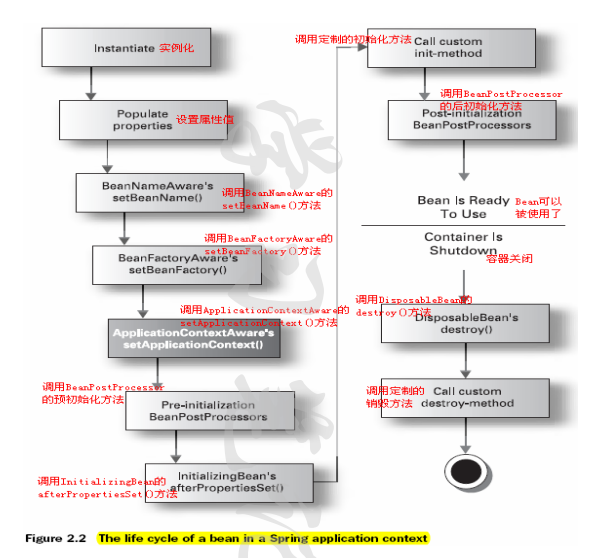
• 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

### Spring Bean 的生命周期

<https://www.cnblogs.com/zrtqsk/p/3735273.html>

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





### Spring IOC 如何实现

<https://www.cnblogs.com/best/p/5727935.html>

<https://blog.csdn.net/a__yes/article/details/52201335>

#### XML配置文件

#### 注解

在类上增加了一个注解Component，在类的开头使用了@Component注解，它可以被Spring容器识别，启动Spring后，会自动把它转成容器管理的Bean。

除了@Component外，Spring提供了3个功能基本和@Component等效的注解，分别对应于用于对DAO，Service，和Controller进行注解。

1：@Repository 用于对DAO实现类进行注解。

2：@Service 用于对业务层注解，但是目前该功能与 @Component 相同。

3：@Constroller用于对控制层注解，但是目前该功能与 @Component 相同。

### Spring中Bean的作用域，默认的是哪一个

<https://www.cnblogs.com/redcool/p/6258973.html>

bean的作用域

singleton---单例 只创建一个对象。

prototype---原型 想创建多少个就创建多少了。

request---针对Web项目，不同的请求创建单独的Bean对象，同一个请求共享一个Bean。

session---针对Web项目，不同的会话创建单独的Bean对象，同一个会话共享一个Bean。

global-session作用域

### 说说 Spring AOP、Spring AOP 实现原理

<https://www.cnblogs.com/CHENJIAO120/p/7080790.html>

Spring AOP有两种实现方式：

基于接口的动态代理(Dynamic Proxy)

基于子类化的CGLIB代理

### 动态代理（CGLib 与 JDK）、优缺点、性能对比、如何选择

使用JDK动态代理，目标类必须实现的某个接口，如果某个类没有实现接口则不能生成代理对象。

Cglib原理是针对目标类生成一个子类，覆盖其中的所有方法，所以目标类和方法不能声明为final类型。

### Spring 事务实现方式、事务的传播机制、默认的事务类别

<https://blog.csdn.net/yuanlaishini2010/article/details/45792069>

<https://www.cnblogs.com/user-peng/p/6703472.html>

<https://blog.csdn.net/qh_java/article/details/51811533>

#### spring事务配置的五种方式

每个Bean都有一个代理

所有Bean共享一个代理基类

使用拦截器

使用tx标签配置的拦截器

全注解

#### spring事务传播属性

在 spring的 TransactionDefinition接口中一共定义了六种事务传播属性：

PROPAGATION\_REQUIRED -- 支持当前事务，如果当前没有事务，就新建一个事务。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 -- 支持当前事务，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，则进行与PROPAGATION\_REQUIRED类似的操作。

前六个策略类似于EJB CMT，第七个（PROPAGATION\_NESTED）是Spring所提供的一个特殊变量。

### Spring 事务底层原理

<https://blog.csdn.net/hanxintong9/article/details/52448205>

#### AOP拦截需要进行事务处理的类

Spring事务处理模块是通过AOP功能来实现声明式事务处理的，具体操作（比如事务实行的配置和读取，事务对象的抽象），用TransactionProxyFactoryBean接口来使用AOP功能，生成proxy代理对象，通过TransactionInterceptor完成对代理方法的拦截，将事务处理的功能编织到拦截的方法中。

### Spring事务失效（事务嵌套），JDK动态代理给Spring事务埋下的坑，可参考《JDK动态代理给Spring事务埋下的坑！》

<https://blog.csdn.net/crystalqy/article/details/79098733>

非通过代理调用。

### 如何自定义注解实现功能

@Retention(RetentionPolicy.RUNTIME)

@Inherited

@Target({ElementType.TYPE, ElementType.METHOD})

public @interface UseDataSource {

String value() default DataSourceConstants.MASTER;

}

<!-- 配置AOP，拦截service方法的执行 -->

<bean id="dynamicDataSourceAspect" class="com.sinitek.common.db.DynamicDataSourceAspect" depends-on="springContextUtil"></bean>

<aop:config>

<aop:pointcut expression ="execution(\* com.sinitek.\*.service..\*.\*(..))" id= "servicePointcut" />

<aop:aspect order="0" ref="dynamicDataSourceAspect">

<aop:around method="setDataSource" pointcut-ref="servicePointcut"></aop:around>

</aop:aspect>

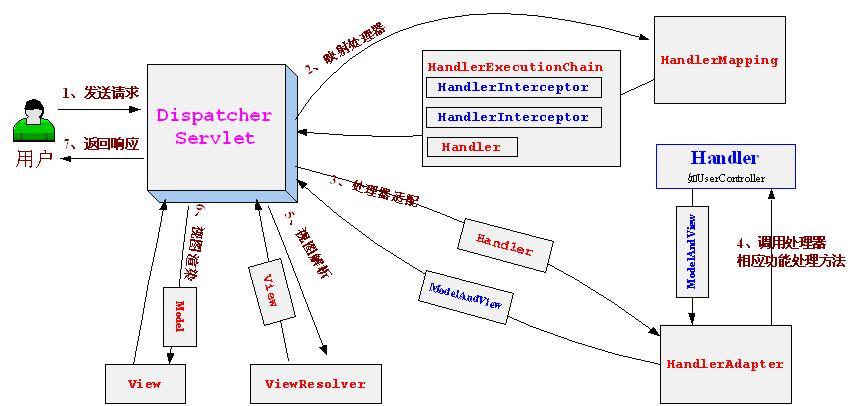
</aop:config>

### Spring MVC 运行流程

<https://blog.csdn.net/liangzi_lucky/article/details/52459378>

<https://www.cnblogs.com/hujiapeng/p/5765636.html>

<https://blog.csdn.net/xiaoxufox/article/details/77196949>



### Spring MVC 启动流程

<https://www.cnblogs.com/mingziday/p/4987058.html>

<https://www.cnblogs.com/RunForLove/p/5688731.html>

Spring MVC启动过程依据这两个配置大致分为两个过程：

1、ContextLoaderListener初始化，实例化IoC容器，并将此容器实例注册到ServletContext中。

1. DispatcherServlet初始化，建立自己的上下文，也注册到ServletContext中。

### Spring 的单例实现原理

<https://blog.csdn.net/u011305680/article/details/79717238>

DefaultSingletonBeanRegistry <--- FactoryBeanRegistrySupport <--- AbstractBeanFactory

### Spring 框架中用到了哪些设计模式

<https://blog.csdn.net/bigtree_3721/article/details/51037547>

### Spring 其他产品（Srping Boot、Spring Cloud、Spring Secuirity、Spring Data、Spring AMQP 等）

### 有没有用到Spring Boot，Spring Boot的认识、原理

<https://blog.csdn.net/xiaobing_122613/article/details/54943448>

<https://www.cnblogs.com/QAZLIU/p/6924643.html>

<https://www.cnblogs.com/moonlightL/p/7891803.html>

### MyBatis的原理

<https://blog.csdn.net/qq_33101675/article/details/78701198>

<https://blog.csdn.net/u014297148/article/details/78696096>

### 可参考《为什么会有Spring》

<https://blog.csdn.net/b644ROfP20z37485O35M/article/details/78967361>

### 可参考《为什么会有Spring AOP》

## Netty

### 为什么选择 Netty

<https://blog.csdn.net/qq_25673113/article/details/54631883>

### 说说业务中，Netty 的使用场景

### 原生的 NIO 在 JDK 1.7 版本存在 epoll bug

<http://www.10tiao.com/html/308/201602/401718035/1.html>

<https://www.cnblogs.com/JAYIT/p/8241634.html>

### 什么是TCP 粘包/拆包

### TCP粘包/拆包的解决办法

<https://yiweifen.com/v-1-49153.html>

定长消息，消息长度是固定的，不够的进行补齐

特殊字符分隔，如FTP协议在包尾添加换行符进行分隔

将消息分为消息投和消息体，消息头中包含消息的总长，如前4个字节表示整个消息的长度（最灵活和常用的方案）

### Netty 线程模型

<https://blog.csdn.net/hao707822882/article/details/39544553>

<https://my.oschina.net/bieber/blog/406799>

<https://blog.csdn.net/xiaolang85/article/details/37873059>

### 说说 Netty 的零拷贝

<https://blog.csdn.net/dc_726/article/details/48978891>

<https://my.oschina.net/plucury/blog/192577>

<http://developer.51cto.com/art/201611/522620.htm>

<https://www.cnblogs.com/xys1228/p/6088805.html>

更多的指的是用户空间的，不是指和内核空间的。

### Netty 内部执行流程

<https://blog.csdn.net/Truong/article/details/62231036>

<https://www.cnblogs.com/NanguoCoffee/archive/2010/12/10/1902491.html>

### Netty 重连实现

<https://segmentfault.com/a/1190000006931568>

ChannelFutureListener

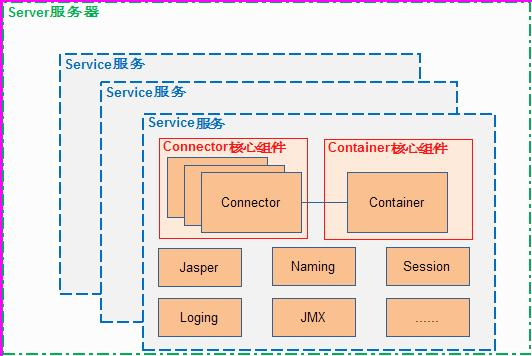
SimpleChannelInboundHandler.channelInactive

IdleStateHandler

## Tomcat

### Tomcat的基础架构（Server、Service、Connector、Container）

<https://blog.csdn.net/cx520forever/article/details/52743166>



### Tomcat如何加载Servlet的

<https://www.cnblogs.com/chenv/p/5869422.html>

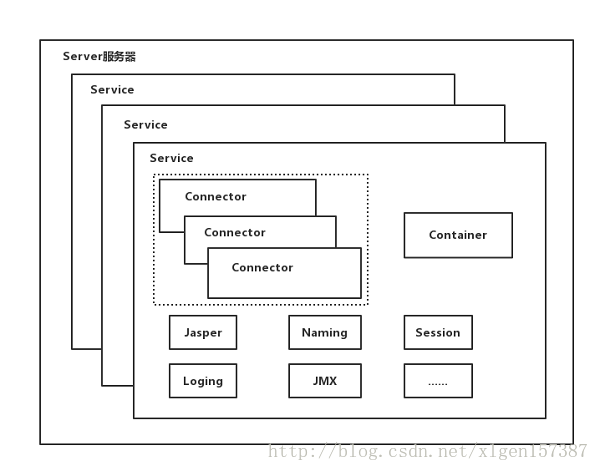
### Pipeline-Valve机制

<https://blog.csdn.net/wangyangzhizhou/article/details/45441355>

<https://www.cnblogs.com/hunter-zyg/articles/4754901.html>

<https://blog.csdn.net/xlgen157387/article/details/79006434>

可参考：《四张图带你了解Tomcat系统架构！》



# 分布式

## Nginx

### 请解释什么是C10K问题或者知道什么是C10K问题吗？

<https://my.oschina.net/xianggao/blog/664275>

<https://segmentfault.com/a/1190000007240744>

### Nginx简介，可参考《Nginx简介》

<https://blog.csdn.net/MrJavaweb/article/details/79635430>

### 正向代理和反向代理

<https://www.cnblogs.com/Anker/p/6056540.html>

正向代理代理客户端，反向代理代理服务器

### Nginx几种常见的负载均衡策略

<https://www.cnblogs.com/andashu/p/6377323.html>

1、轮询（默认）

每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。

upstream backserver {

server 192.168.0.14;

server 192.168.0.15;

}

2、指定权重

指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况。

upstream backserver {

server 192.168.0.14 weight=10;

server 192.168.0.15 weight=10;

}

3、IP绑定 ip\_hash

每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器，可以解决session的问题。

upstream backserver {

ip\_hash;

server 192.168.0.14:88;

server 192.168.0.15:80;

}

4、fair（第三方）

按后端服务器的响应时间来分配请求，响应时间短的优先分配。

upstream backserver {

server server1;

server server2;

fair;

}

5、url\_hash（第三方）

按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，后端服务器为缓存时比较有效。

upstream backserver {

server squid1:3128;

server squid2:3128;

hash $request\_uri;

hash\_method crc32;

}

### Nginx服务器上的Master和Worker进程分别是什么

<https://blog.csdn.net/yusiguyuan/article/details/40924415>

master进程

监控进程充当整个进程组与用户的交互接口，同时对进程进行监护。它不需要处理网络事件，不负责业务的执行，只会通过管理worker进程来实现重启服务、平滑升级、更换日志文件、配置文件实时生效等功能。

worker进程

worker进程的主要任务是完成具体的任务逻辑。其主要关注点是与客户端或后端真实服务器（此时nginx作为中间代理）之间的数据可读/可写等I/O交互事件，所以工作进程的阻塞点是在像select()、epoll\_wait()等这样的I/O多路复用函数调用处，以等待发生数据可读/写事件。当然也可能被新收到的进程信号中断。

<https://my.oschina.net/tickbh/blog/1796292>

由work进程通过一个一把锁保证只有一个进程accept

### 使用“反向代理服务器”的优点是什么?

## 分布式其他

### 谈谈业务中使用分布式的场景

### Session 分布式方案

<https://blog.csdn.net/u010028869/article/details/50773174>

粘性session

服务器session复制

session共享机制

session持久化到数据库

terracotta实现session复制

### Session 分布式处理

<https://blog.csdn.net/yangrui0070/article/details/60572777>

<https://www.cnblogs.com/cxrz/p/8529587.html>

### 分布式锁的应用场景、分布式锁的产生原因、基本概念

<https://blog.csdn.net/daiyudong2020/article/details/51660994>

<http://www.weizijun.cn/2016/03/17/%E8%81%8A%E4%B8%80%E8%81%8A%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81%E7%9A%84%E8%AE%BE%E8%AE%A1/>

### 分布式锁的常见解决方案

<https://www.cnblogs.com/yuyutianxia/p/7149363.html>

数据库锁（唯一ID）

Redis的setnx锁

zookeeper实现锁的方式是客户端一起竞争写某条数据，比如/path/lock，只有第一个客户端能写入成功，其他的客户端都会写入失败。写入成功的客户端就获得了锁，写入失败的客户端，注册watch事件，等待锁的释放，从而继续竞争该锁。

### 分布式事务的常见解决方案

<https://www.zhihu.com/question/64921387/answer/225784480>

<https://blog.csdn.net/mine_song/article/details/64118963>

<https://blog.csdn.net/congyihao/article/details/70195154>

<https://blog.csdn.net/congyihao/article/details/70195154>

<https://blog.csdn.net/qq_17805795/article/details/54910536>

两阶段提交（2PC, Two-phase Commit）方案

eBay 事件队列方案（提交事务的同时发送消息）

异步确保型（通过将一系列同步的事务操作变为基于消息执行的异步操作, 避免了分布式事务中的同步阻塞操作的影响.）

TCC 补偿模式（Try、Confirm和Cancel三个操作）

缓存数据最终一致性

### 集群与负载均衡的算法与实现

<https://blog.csdn.net/ioy84737634/article/details/44916241>

#### LVS 调度算法

总共八种方法， 最脑残暴力的就是直接平均轮叫。

##### 轮叫调度

调度器通过外部请求的顺序轮流的分配到集群中的真实服务器上，对每台服务器都是均等的。但是这样调度器不会考虑服务器上实际的连接数和系统负载，导致服务器处理请求慢，系统负载增大。

##### 加权轮叫

调度器通过一个算法根据真实服务器的不同处理能力来分配访问请求，这样可以保证服务器的处理能力。

##### 最少连接

调度器将访问请求自动的分配到已建立连接最少的服务器上，如果在集群中每台服务器的性能差不多的话，则这种算法可以较好的均衡负载。

##### 加权最少连接

主要用于集群中服务器性能差异大的情况下，调度器可以优化负载性能，具有较高权值的服务器可以将承受较大的活动连接。

##### 基于局部性的最少连接

主要是针对目标IP地址的负载均衡，将请求的目标IP地址找到离其最近的服务器进行使用，如果服务器不存在或者满载的话，就会继续寻找下一个服务器。

##### 带复制的基于局部性的最少连接

主要是针对目标IP地址的负载均衡，根据请求的目标IP地址找出该地址所对应的服务器，如果服务器不存在或者满载的话，就会继续寻找下一个服务器。当服务器有一段时间没有被修改，则会从最忙的服务器组中删除。

##### 目标地址散列

根据请求的目标IP地址从静态分配的散列表中超出对应的服务器，如果找到可用的服务器且没有满载，则返回空。

##### 源地址散列

根据请求的源IP地址从静态分配的散列表中超出对应的服务器，如果找到可用的服务器且没有满载，则返回空。

### 说说分库与分表设计，可参考《数据库分库分表策略的具体实现方案》

<https://blog.csdn.net/xlgen157387/article/details/53976153>

### 分库与分表带来的分布式困境与应对之策

<https://blog.csdn.net/jiangpingjiangping/article/details/78069480>

<http://blog.720ui.com/2017/mysql_core_09_multi_db_table2/>

数据迁移与扩容问题

表关联问题

分页与排序问题

分布式事务问题

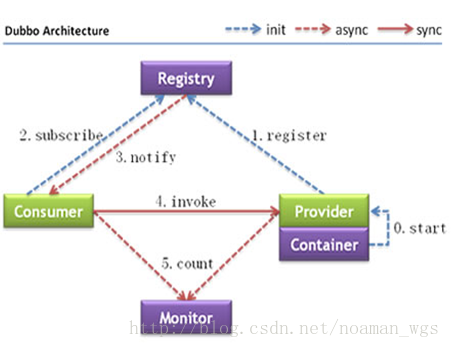
分布式全局唯一ID

## Dubbo

### 什么是Dubbo，可参考《Dubbo入门》

<https://blog.csdn.net/noaman_wgs/article/details/70214612/>

<https://blog.csdn.net/xia744510124/article/details/78143731>



### 什么是RPC、如何实现RPC、RPC 的实现原理，可参考《基于HTTP的RPC实现》

<https://blog.csdn.net/xlgen157387/article/details/53543009>

### Dubbo中的SPI是什么概念

Service Provider Interface

<https://www.cnblogs.com/senlinyang/p/8612458.html>

<https://blog.csdn.net/qiangcai/article/details/77750541>

java.util.ServiceLoader;

在META-INF文件夹下创建了services文件夹，并在services文件夹下创建了一个文件，文件名以接口的全限定名来命名，我这里是spi.DubboService。然后在这个文件中填入这个接口的两个实现类，类中间以换行隔开。

### Dubbo的基本原理、执行流程

<http://www.mamicode.com/info-detail-1989236.html>

<https://blog.csdn.net/hzzhoushaoyu/article/details/43273099>

# 微服务

## 微服务

### 前后端分离是如何做的？

<https://blog.csdn.net/wangji5850/article/details/51131688>

<https://blog.csdn.net/weixin_38639882/article/details/79296583>

<https://blog.csdn.net/github_26672553/article/details/51864112>

### 微服务哪些框架

<https://blog.csdn.net/qiansg123/article/details/80130182>

Spring boot

### Spring Could的常见组件有哪些？可参考《Spring Cloud概述》

<https://blog.csdn.net/sanyaoxu_2/article/details/80483519>

<https://springcloud.cc/spring-cloud-dalston.html#_circuit_breaker_hystrix_clients>

### 领域驱动有了解吗？什么是领域驱动模型？充血模型、贫血模型

<https://kb.cnblogs.com/page/520746/>

贫血模型是指使用的领域对象中只有setter和getter方法（POJO），所有的业务逻辑都不包含在领域对象中而是放在业务逻辑层。

充血模型将大多数业务逻辑和持久化放在领域对象中，业务逻辑只是完成对业务逻辑的封装、事务和权限等的处理。

### JWT有了解吗，什么是JWT，可参考《前后端分离利器之JWT》

<https://blog.csdn.net/bntx2jsqfehy7/article/details/79224042>

Json web token (JWT)

<https://www.jianshu.com/p/180a870a308a>

Header.Payload.Signature

### 你怎么理解 RESTful

<https://www.cnblogs.com/binlin1987/p/6971808.html>

REST的名称"表现层状态转化"中，省略了主语。"表现层"其实指的是"资源"（Resources）的"表现层"。

### 说说如何设计一个良好的 API

<http://www.jdon.com/48452>

<https://blog.csdn.net/shog808/article/details/79932968>

### 如何理解 RESTful API 的幂等性

<http://blog.720ui.com/2016/restful_idempotent/>

### 如何保证接口的幂等性

<https://blog.csdn.net/wangyan9110/article/details/70953273>

查询是幂等的。

PUT去重检查

### 说说 CAP 定理、BASE 理论

<https://blog.csdn.net/sunxinhere/article/details/7936485>

<https://blog.csdn.net/xiaqunfeng123/article/details/51688974>

CAP：一致性（Consistency）、可用性（Availability）、分区容忍性（Partition Tolerance）

BASE全称是BasicallyAvailable（基本可用）, Soft-state（软状态/柔性事务）, Eventually Consistent（最终一致性）

### 怎么考虑数据一致性问题

<https://www.cnblogs.com/soundcode/p/5590710.html>

### 说说最终一致性的实现方案

<https://blog.csdn.net/zxl315/article/details/53433707>

<https://www.cnblogs.com/lzyGod/p/5558474.html>

<https://segmentfault.com/a/1190000011479826>

<https://blog.csdn.net/hu_zhiting/article/details/77164138>

<http://rickgong.iteye.com/blog/2369662>

将需要分布式处理的任务通过消息日志的方式来异步执行

将主要修改操作以及更新用户表的消息放在一个本地事务来完成。同时为了避免重复消费用户表消息带来的问题，达到多次重试的幂等性，增加一个更新记录表 updates\_applied 来记录已经处理过的消息。

消息最大努力交付（上游应用，发消息到 MQ 队列。下游应用（例如短信服务、邮件服务），接受请求，并返回通知结果。最大努力通知服务，监听消息队列，将消息存储到数据库中，并按照通知规则调用下游应用的发送通知接口。）

可靠消息最终一致性方案（上游应用，执行业务并发送 MQ 消息。可靠消息服务和 MQ 消息组件，协调上下游消息的传递，并确保上下游数据的一致性。下游应用，监听 MQ 的消息并执行自身业务。）

TCC

### 微服务的优缺点，可参考《微服务批判》

### 微服务与 SOA 的区别

<https://www.cnblogs.com/ynuo/p/5913955.html>

微服务与SOA相比，更强调分布式系统的特性，比如横向伸缩性，服务发现，负载均衡，故障转移，高可用。互联网开发对服务治理提出了更多的要求，比如多版本，比如灰度升级，比如服务降级，比如分布式跟踪，这些都是在SOA实践中重视不够的。

### 如何拆分服务、水平分割、垂直分割

<https://blog.csdn.net/Allen202/article/details/75174829>

<https://blog.csdn.net/u012422829/article/details/68951579?utm_source=itdadao&utm_medium=referral>

<https://blog.csdn.net/zhangweiwei2020/article/details/78018876>

根据业务能力拆分

根据子域拆分

### 如何应对微服务的链式调用异常

使用超时策略

使用熔断器机制。

### 如何快速追踪与定位问题

<http://www.infoq.com/cn/articles/how-to-realize-distributed-tracking/>

<https://www.cnblogs.com/rongfengliang/p/6209301.html>

TraceId、parentSpanId、spanId

### 如何保证微服务的安全、认证

<https://blog.csdn.net/u011676417/article/details/73481102/>

单点登录（SSO）

分布式 Session 方案

客户端 Token 方案（ JSON Web Tokens（JWT））

客户端 Token 与 API 网关结合

OAuth 2.0

## 安全问题

### 如何防范常见的Web攻击、如何方式SQL注入

<https://www.cnblogs.com/csniper/p/5802202.html>

<https://blog.csdn.net/stpeace/article/details/53512283>

1、SQL注入攻击

2、跨站脚本攻击 - XSS

3、跨站伪造请求攻击 - CSRF

4、文件上传漏洞攻击

5、分布式拒绝服务攻击 - DDOS

验证 HTTP Referer 字段

### 服务端通信安全攻防

<http://blog.720ui.com/2016/security_data_transmission/>

### HTTPS原理剖析、降级攻击、HTTP与HTTPS的对比

<http://blog.jobbole.com/106792/>

<http://blog.720ui.com/2016/security_https_tls/>

HTTPS协议在应用层的http协议和以及传输层TCP协议之间的会话层加入了 SSL/TLS协议，用作加密数据包。会话层的这个加密协议，在真正数据传输之前，进行握手通信，握手的信息交换主要包括：双方确认加密协议的版本，确认身份（其中浏览器客户端的身份有时是不需要确认的），确认传输真正数据的加密密码（对称加密）。

## 性能优化

### 性能指标有哪些

<https://www.cnblogs.com/zxf330301/p/6354519.html?utm_source=itdadao&utm_medium=referral>

资源指标

CPU使用率：指用户进程与系统进程消耗的CPU时间百分比，长时间情况下，一般可接受上限不超过85%。

内存利用率：内存利用率=（1-空闲内存/总内存大小）\*100%，一般至少有10%可用内存，内存使用率可接受上限为85%。

磁盘I/O: 磁盘主要用于存取数据，因此当说到IO操作的时候，就会存在两种相对应的操作，存数据的时候对应的是写IO操作，取数据的时候对应的是是读IO操作，一般使用% Disk Time（磁盘用于读写操作所占用的时间百分比）度量磁盘读写性能。

网络带宽：一般使用计数器Bytes Total/sec来度量，Bytes Total/sec表示为发送和接收字节的速率，包括帧字符在内。判断网络连接速度是否是瓶颈，可以用该计数器的值和目前网络的带宽比较。

系统指标：

并发用户数：某一物理时刻同时向系统提交请求的用户数。

在线用户数：某段时间内访问系统的用户数，这些用户并不一定同时向系统提交请求。

平均响应时间：系统处理事务的响应时间的平均值。事务的响应时间是从客户端提交访问请求到客户端接收到服务器响应所消耗的时间。对于系统快速响应类页面，一般响应时间为3秒左右。

事务成功率：性能测试中，定义事务用于度量一个或者多个业务流程的性能指标，如用户登录、保存订单、提交订单操作均可定义为事务，如下图所示：

### 如何发现性能瓶颈

<https://blog.csdn.net/beijiguangyong/article/details/17684797>

监控、定位

### 性能调优的常见手段

<https://www.cnblogs.com/ajianbeyourself/p/6132546.html>

优先级大小

硬件

数据库（SQL调优）

缓存

JVM调优

### 说说你在项目中如何进行性能调优

# 其他

## 设计能力

### 说说你在项目中使用过的UML图

### 你如何考虑组件化、服务化、系统拆分

不同业务模块、基础组件

### 秒杀场景如何设计

<https://blog.csdn.net/suifeng3051/article/details/52607544>

<https://blog.csdn.net/qq_22912803/article/details/56842033>

限流

削峰

异步

缓存

可扩展（临时增加机器）

前端（前端页面静态化、重复提交、限制提交频率）

后端（限制用户访问频率、Redis的RPUSH和LPOP）

<https://blog.csdn.net/bntx2jsqfehy7/article/details/79407907>

可参考：《秒杀系统的技术挑战、应对策略以及架构设计总结一二！》

独立部署、页面静态化（js来控制是否开始秒杀）、租借带宽、下单URL随机生成、

## 业务工程

### 说说你的开发流程、如何进行自动化部署的

<https://blog.csdn.net/fanyun7654/article/details/20690787>

操作系统一致

代码编译自动化

### 你和团队是如何沟通的

### 你如何进行代码评审

<https://www.cnblogs.com/yzycoder/p/6873601.html>

Checklist（代码规范）

### 说说你对技术与业务的理解

技术能力、主动态度、团队合作

### 说说你在项目中遇到感觉最难Bug，是如何解决的

### 介绍一下工作中的一个你认为最有价值的项目，以及在这个过程中的角色、解决的问题、你觉得你们项目还有哪些不足的地方

### 软实力

### 说说你的优缺点、亮点

### 说说你最近在看什么书、什么博客、在研究什么新技术、再看那些开源项目的源代码

### 说说你觉得最有意义的技术书籍

### 工作之余做什么事情、平时是如何学习的，怎样提升自己的能力

### 说说个人发展方向方面的思考

### 说说你认为的服务端开发工程师应该具备哪些能力

### 说说你认为的架构师是什么样的，架构师主要做什么

### 如何看待加班的问题