# 第一篇 J2SE

## Base

### 1.1.1 Java发展

JDK 1.1

1997年发布。

* 反射
* JDBC；
* 支持内部类；
* 引入Java Bean；
* 引入RMI（Remote Method Invocation）；

JDK 1.2

1998年发布

* Collection
* 引入Java 插件；
* 对字符串常量做内存映射；
* 引入JIT（Just In Time）编译器；
* 引入对打包的Java文件进行数字签名；
* 引入控制授权访问系统资源的策略工具；
* 引入JFC（Java Foundation Classes），包括Swing 1.0、拖放和Java 2D类库；

JDK 1.4

2002年发布

* NIO
* XML处理；
* Java打印服务；
* 引入Logging API；
* 引入Java Web Start；
* 引入断言Assert；

引入Preferences API；

* 引入链式异常处理；
* 支持IPv6；
* 支持正则表达式；
* 引入Image I/O slot machine API。

JDK 5

2004年发布

[Reference](https://sakuraffy.github.io/java_jdk5.0/)

* 泛型
* concurrent包
* 枚举
* 元数据（Metadata）
* foreach循环
* 自动拆箱装箱
* 静态导入
* 可变参数 add(int... arr)
* 内省

JDK6

2006年发布

* 支持脚本语言

JDK7

2011年发布

* 支持动态语言
* 异步IO（NIO 2）
* Lambda的实现

JDK 8

2014年发布

* Lambda表达式
* HashMap增加红黑树查找
* Optional防NPE
* Pipelines和Streams
* Date和Time API
* Default方法
* Type注解
* Nashhorn JavaScript引擎
* 并发计数器
* Parallel操作
* 移除PermGen Error
* TLS SNIJava 8的内存分代改进

JDK9

2017年发布

* **模块系统**：模块是一个包的容器，Java 9 最大的变化之一是引入了模块系统（Jigsaw 项目）。
* **HTTP 2 客户端**：HTTP/2标准是HTTP协议的最新版本
* **REPL (JShell)**：交互式编程环境。
* **响应式流（Reactive Streams) API**: 支持响应式编程。
* **集合工厂方法**
* **私有接口方法**：在接口中使用private私有方法。
* **改进 try-with-resource**
* **改进的 CompletableFuture API** ：
* **进程 API**: 改进的 API 来控制和管理操作系统进程。
* **改进的 Stream API**
* **改进的 Javadoc**
* **改进钻石操作符(Diamond Operator)**：匿名类可以使用钻石操作符(Diamond Operator)。

Java 10

局部变量类型推断、G1 的并行 Full GC、ThreadLocal 握手机制

Java 11

ZGC、Epsilon、增强 var

### 1.1.2 JDK

TODO3JDK&JRE&JVM

和Google的Android关系https://www.zhihu.com/question/270271649

### 1.1.3 接口&抽象

接口：体现的是一种规范和实现分离的设计哲学。

设计层面

抽象类是对类的抽象，包括属性、行为，子类继承，描述is a的关系。而接口是对行为的抽象，是like a，约束类的行为。

(实际应用中接口优于抽象类：接口更加抽象，所以更稳定?)。

语法层面

1.抽象类 可以有自己的数据成员，也可以有非abstract的成员方法，而接口中只能存在public abstract 方法；

2.一个类只能使用一次继承关系而一个类却可以实现多个interface

语言层面

不同语言设计的哲学不同，就拿继承来说：

其实C++就支持多继承（这也是java吐槽的），这么多C++代码，也没见出多大乱子给，程序员带来多大痛苦，python多继承，也活的好好的，多少人热爱它。很多现代语言，比如swift，还有extension这种玩意儿， 动态的给一个类或者一个结构体挂个新的方法，也被很多程序员所接受。 Java描述抽象类，就是继承，描述is a的关系，接口是like a，约束类的行为。

### 1.1.4 内部类

1.完善多继承。

可以直接使用其外部类的成员变量以及成员函数，达到一个继承的效果，再加上自身继承基类来达到一个多重继承的效果

1. **public** **class** OuterClass {
2. //内部类
3. **class** InnerClass **extends** CommonClass{
5. **public** **void** enhanceMethod1(){
6. System.out.println("before");
7. **super**.method1();//调用父类commonClass的函数
8. method();//调用外部类outerClass的函数
9. System.out.println("after");
10. };
11. }
12. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
13. **public** **void**  method(){
14. System.out.println("method函数");
15. }
16. }

2.闭包

定义在方法中的成员内部类?。辅助解决复杂问题，又不希望此类公用。

3.匿名内部类

匿名类：直接new 接口，没有名字的类。

匿名内部类：可以访问外层类的字段的匿名类。

使用场景：实现类只使用一次，不用去新建类，简化代码编写

比如Thread，自定义的接口

1. **new** Thread(**new** Runnable() {
2. @Override
3. **public** **void** run() {
4. //statement
5. }
6. }).start();

事件驱动

swing使用大量的内部类

### 1.1.5 反射

1.相比new，反射可以根据类名动态加载类(required)，是另一种加载类的方法。

2.Runtime阶段根据类名动态获取类信息并能够调用类方法。

* 使用场景：反射机制在做基础框架的时候非常有用。而一般应用层面自己使用场景很少。典型的除了Hibernate之外，还有Spring也用到很多反射机制。经典的就是根据配置加载类。
* 增加动态语言的特性

框架为什么用反射？

动态加载类

需要延迟到runtime阶段动态加载。比如根据配置加载类。

runtime阶段需要访问类的私有属性

Java之动态语言特性

动态语言：运行时可改变其结构：1.改变变量类型2.添加或删除函数。比如JavaScript, Ruby,Python都属于动态语言，Java和C、C++一样都不属于动态语言。

* 反射让Java有半动态的语言特性（但Java通过反射机制，可以在程序运行时加载，探知和使用编译期间完全未知的类，并且可以生成相关类对象实例，从而可以调用其方法或则改变某个属性值）。
* lambda让Java有了函数式编程语言的特性。

静态编译 VS 动态编译

静态编译：一次性编译。在编译的时候把你所有的模块都编译进去。比如编译成一个程序exe，运行时一次性全部加载。

动态编译：按需编译。程序在运行的时候，用到那个模块就编译哪个模块。C: 按模块编译成ddl按需加载。Java: 反射加载类。

Class.forName和classloader的区别

Class.forName(xxx);的作用是要求JVM查找并加载指定的类。

反射效率低：需要读取字节码到内存然后分配内存生成类实例最后执行。

内省（Introspector）

### 1.1.6 泛型

TODO3

Assert

TODO3

transient

TODO2

## 1.2 Data Type

### Object

equals

1. **public** **boolean** equals(Object obj) { **return** (**this** == obj);}
2. **public** **native** **int** hashCode();

equals :“==”比较两个对象的的内存地址，可见默认的equals判断是不是同一个对象。

hashCode: 默认根据对象地址生成一个整数数值(散列存储结构中确定对象的存储地址)；

object重写equals为何要重写hashCode

object的hashCode值默认根据对象地址生成，Set和Map添加对象时都是先根据hashCode是不是一样，再判定两个对象是否equals;可能导致若两个对象equals相等hashCode不同，集合出现重复元素（required）重写hashCode保证两个equals，hashCode相同。

从官方文档的定义，可以得到equals与hashCode间的关系：

(想象往HashMap添加对象)

1、两个对象equals，hashCode一定要相同。

2、两个对象的hashCode相同，它们并不一定equals。

目的：保证集合元素不能重复。

wait ()/notify ()/notifyAll ()



Warning：线程wait后被notify，要重新获得lock才能从中断处继续执行。

### String

[Reference](http://www.hollischuang.com/archives/2517)

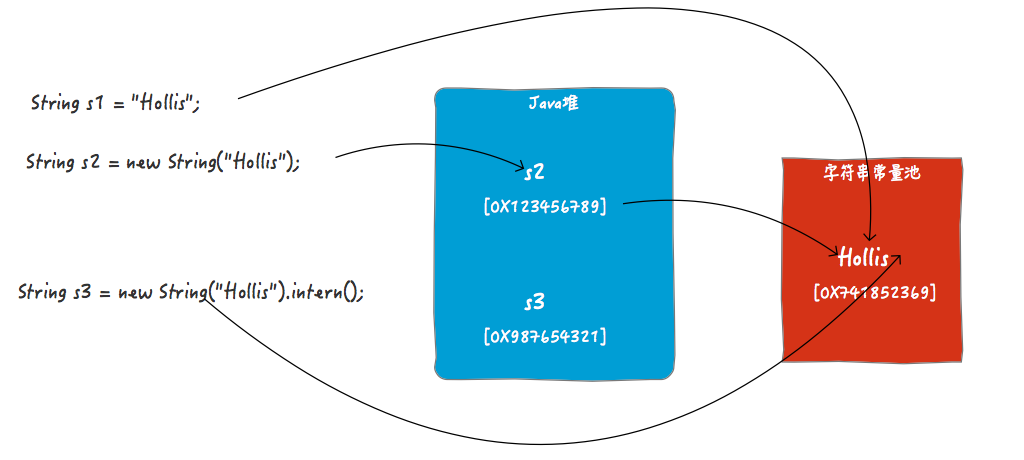
M：String s = new String("hollis")定义了几个对象?

A1: 若常量池中已经存在”hollis”，则直接引用，此时只会在Heap中创建一个对象，如果常量池中不存在”hollis”，则先创建后引用，也就是有两个。

运行时常量池

存储编译器生成各种字面量和符号引用,String s1="ndong"，字面量：ndong，符号引用：s1。

创建对象问题



String s1=new String("Hollis")

编译期：符号引用s2和字面量Hollis会被加入类文件的常量池，在类加载阶段进入常量池。

运行期：Heap中创建一个String对象，它的成员变量char[] 指向常量池中对象" Hollis "

s1,s2,s3均保持在stack的本地变量中

intern()

运行期向常量池添加字符串对象，并返回其地址。

String s3 = new String("Hollis").intern()中intern()是多余的，new在运行期已经添加。

使用场景：

#### Immutable



数组value是也是对象，引用地址是可变的，可见String的不可变性在于底层的实现，而不是一个final。

**目的**

设计成immutable的类，可以像普通类型一样赋值，而不是普通的类。

**好处**

主要是为了“效率” 和 “安全性” 的缘故。

效率：对象的不可变性，所以可以共享。节省内存空间。

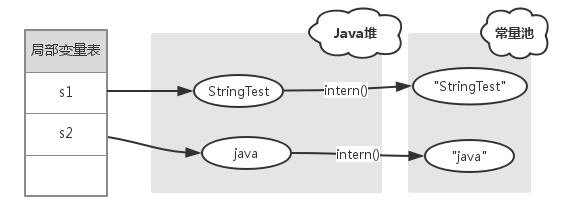
安全性：String作为对象，可以像基本类型值传递，而不改变对象的值。

不仅String 是final的。 Long, Double, Integer 之类的全都是final的

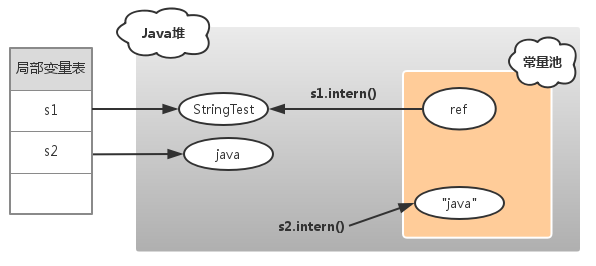
#### 常量池

[Reference](https://www.jianshu.com/p/0d1c003d2ff5)

1. **class** Test {
2. **public** **static** **void** main(String args[]) {
3. String s1 = **new** StringBuilder().append("String").append("Test").toString(); System.out.println(s1.intern() == s1);
5. String s2 = **new** StringBuilder().append("ja").append("va").toString();
6. System.out.println(s2.intern() == s2);
7. }
8. }
9. **JDK6的执行结果：false, false**



1. **JDK7的执行结果：true false**



String str=”a”



ldc先在常量池创建对象

String str=new String(“a”)



ldc “str”过程

触发oop StringTable::intern

“+=”的重载

“+=” 是Java中仅有的两个重载过的操作符,而java并不允许我们重载任何操作符。

**对+的重载**

加号被转换为StringBuffer的append()方法来操作,最后toString生成新的对象。



String.valueOf和Integer.toString的区别、

### Integer

包装类：

Java的8种基本类型都有对应的包装类。包装类与基本类型的转换：

**自动拆装箱**



装箱:Integer.valueOf()

拆箱：Integer.intValue()

需要注意拆箱时可能发生NPE？

valueOf

缓存机制



可见在-128~128范围



Integer.MAX\_VALUE

nteger是4位，4\*8=32,去掉一个符号位，-2^31~2^31-1)

### 数组

final数组

数组也是对象，引用arr1的地址不能改变，内容可改变



数组复制

**浅拷贝**: 不进行递归拷贝，直接拷贝内存本身的值，所以产生的结果是基本数据类型是值拷贝，对象只是引用拷贝

Arrays.copyOf 底层调用System.arraycopy，浅拷贝

**深拷贝**

Object.clone 对于对象而言，它是深拷贝，但是对于数组而言，它是浅拷贝

### 变量

**局部变量**：方法中的变量。分配在stack

**类变量**：方法之外的变量，用 static 修饰。分配在方法区。

**实例变量**：方法之外的变量，不过没有 static 修饰。分配在堆

**成员变量**（全局变量）：包括实例变量和类变量

总结: 方法内定义的变量和引用全部在栈中，方法外定义的**实例变量全部在堆**中，静态变量在方法区。

**常量**

和变量相对，不能更改。在Java中，指final修饰的变量。

好处：

* 使用final关键字，JVM会对方法、变量及类进行优化
* 提高了性能。JVM和Java应用都会缓存final变量。
* 在多线程环境下进行共享，而不需要额外的同步开销。

**常量池**

实现常量池技术的基本类型包装类：Integer, Long，String

### 引用类型

StrongReference

强引用。普遍的引用。

SoftReference

软引用。内存不足时回收，可用来实现内存敏感的高速缓存

1. String str=**new** String("abc");
2. SoftReference<String> softRef=**new** SoftReference<String>(str);

WeakReference

弱引用。不管当前内存空间足够与否，都会回收它的内存

PhantomReference

虚引用。在任何时候都可能被垃圾回收器回收。主要用来跟踪对象被垃圾回收器回收的活动，虚引用必须和引用队列 （ReferenceQueue）联合使用

### final

**final Class:**类不能被继承。方法不能rewrite

**final Method:**方法不能rewrite

**final Variable** 修饰的基本类型的变量，初始化后不能更改。修饰引用类型不能执行另一个对象，但对象内容可以更改。

修饰基本类型



## 1.3 Collection

### 1.2.1 概述



**Collection VS Array**

数组为定长，Collection均可动态增长。

Queue和tree的具体内容见算法章节。

### 1.2.2 List

元素保持特定的顺序

#### ArrayList

基于动态数组。

实现原理

1. **public** **class** ArrayList<E> {
2. **private** **static** **final** **int** DEFAULT\_CAPACITY = 10;
3. **private** **transient** Object[] elementData;
4. **private** **int** size;
5. }

基于数组实现，可动态的扩展和缩减

DEFAULT\_CAPACITY = 10

Add

1. **public** **boolean** add(E e) {
2. ensureCapacityInternal(size + 1);  // Increments modCount!!
3. elementData[size++] = e;
4. **return** **true**;
5. }

扩容

1. **private** **void** grow(**int** minCapacity) {
2. // overflow-conscious code
3. **int** oldCapacity = elementData.length;
4. **int** newCapacity = oldCapacity + (oldCapacity >> 1);
5. **if** (newCapacity - minCapacity < 0)
6. newCapacity = minCapacity;
7. **if** (newCapacity - MAX\_ARRAY\_SIZE > 0)
8. newCapacity = hugeCapacity(minCapacity);
9. // minCapacity is usually close to size, so this is a win:
10. elementData = Arrays.copyOf(elementData, newCapacity);
11. }

minCapacity = size +1

Arrays.copyof

其使用的是内存复制，省去了大量的数组寻址访问等时间

#### LinkedList

基于链表的数据结构

ArrayList VS LinkedList

* Get：ArrayList要优于LinkedList，O(1)，因为LinkedList要移动指针；
* Add/remove：一般LinkedList要比Array快（Array索引结构，要移动数据）。但ArrayList基于动态数据在尾部添加，不能明确说明谁快谁慢，看插入位置。

### 1.2.3 Set

HashSet

元素不重复，无顺序。

原理：基于 HashMap 实现的，数据存在key，value都是一个空对象。

LinkedHashSet

维护插入顺序，性能略低于HashSet

### 1.2.3 Map

[Reference](https://www.jianshu.com/p/e2f75c8cce01)

每个元素为key-value形式，支持key,value为null

Summary

* HashTable: 线程安全，方法都加了synchronzed。全局只有一把锁，同时只能一个操作。
* LinkedHashMap：HashMap的子类，保存了插入顺序。
* ConcurrentHashMap：引入“分段锁”概念, 把Map分成了N个Segment，每个段使用一把锁，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中。
* TreeMap: 对插入记录按key排序。可自定义Comparable。

#### HashMap

底层是一个数组结构，数组中的每一项又是一个链表(数组+链表)。非线程安全。

put

1. **public** V put(K key, V value){
2. **int** hash= hashcode（key）;
3. Entry[] list=[hash %table.length];//1.求余取对应位置链表
4. **for**(Entry e:list){
5. //2.若存在（同一链表hash不一定相同）
6. **if**(e.hash.equal(hash)&&e.key.equal(key)){）
7. e.value=value;
8. **return**;
9. }
10. }
11. addToLinkedList(key,value);//3.加到链表后部
12. }

扩容机制

每次put元素是判断。

1. **if** size>capacity \* loadFactor（loadFactor=0.75）
2. 用一个新的数组代替已有的容量小的数组
3. newCapacity=2 \* table.length
4. Entry[] newTable = new Entry[newCapacity];//1.新建
5. transfer(newTable)//2.循环table，根据hash插入newTable
6. end **if**;

缺省大小 DEFAULT\_INITIAL\_CAPACITY =16

碰撞冲突

对应位置链表的next

HashMap冲突厉害，最差的情况可能所有数据都在一个链表中。查找复杂度：O（n）。

如何优化?从O（n）提升到log（n）咯，用二叉排序树的思路说了一通

[Reference](https://www.jianshu.com/p/e2f75c8cce01)

JDK8增加了红黑树的数据结构，使得在bucket里面查找数据的复杂度从O(n)降到O(logn)，当然还有一些其他的优化，比如resize的优化等

get



#### ConcurrentHashMap

[Reference](https://mp.weixin.qq.com/s?__biz=MzI4NDY5Mjc1Mg==&mid=2247485377&idx=1&sn=1c3a232f403f2182f2fdbebf0f268ca9&chksm=ebf6dfbedc8156a85d6e51f6a2cafb13a0c347292030661769c2ca56576c74ef134536261cc7&mpshare=1&scene=1&srcid=0801r1glChVdmW4KKsasq2zD&pass_ticket=1XCbcUFwF3JzJcuXceRdo3OtsbWJJXviPxO6Y4JmdM4NWqQvkGbIctEUzRz7F9Ls#rd)

TODO

## IO

流

字符流、字节流、输入流、输出流、

### 1.4.1 Linux IO 模型

Linux内核将所有外部设备都可以看做一个文件来操作。服务器端的I/O有两种情况：1.网络的I/O；2.对文件(设备)的I/O。

一个IO的read操作分成了两个阶段：

1）用户进程发起system call，数据会被拷贝到内核的缓冲区中。

2）数据从Kernel的缓冲区拷贝到应用程序的地址空间。

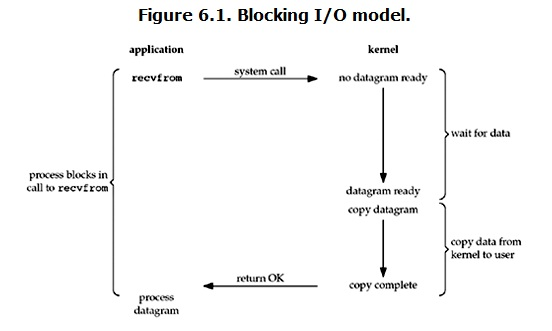
缺点：多次数据拷贝操作内存开销是非常大的。

正因为这两个阶段，linux系统产生了下面五种网络模式的方案。

* blocking IO:进程主动block数据准备好了
* Non-blocking IO: 用户进程不断地主动轮询kernel。
* IO multiplexing：select/epoll同时处理多个文件描述符。
* asynchronous IO：异步 I/O
* signal driven IO：信号驱动 I/O，不常用

[Reference](https://segmentfault.com/a/1190000003063859)

Blocking IO



用户进程调用read操作，将数据拷贝到系统缓存，再从kernel拷贝到用户进程。

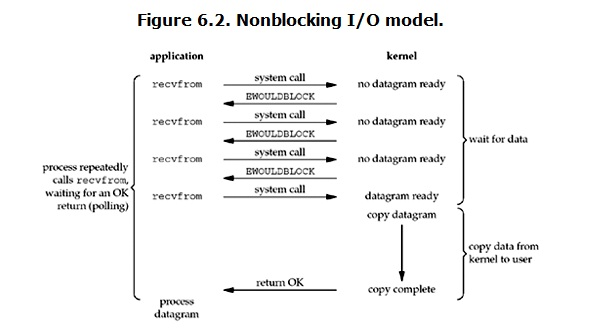
。如采用多线程，将创建大量线程，并有大量的线程切换。

recv/send/recvfrom

1. ssize\_t recv(**int** sockfd, **void** \*buff, **size\_t** nbytes, **int** flags);
2. ssize\_t send(**int** sockfd, **const** **void** \*buff, **size\_t** nbytes, **int** flags);

sockfd：发送端的fd，比如服务器的角度则是客户端，buff ：存放接收的数据，send()/recv()用于TCP，sendto/recvfrom（）用于UDP

Non-blocking IO



若kernel没准备好数据，用户进程不block，而是返回error。用户进程不断主动询问，效率较低，要结合IO multiplexing

Java非阻塞原理

在Java中要使用非阻塞

1. SocketChannel socketChannel = SocketChannel.open();
2. socketChannel.configureBlocking(**false**);

在JDK源码中实现

1. **static** **int** configureBlocking(**int** fd, jboolean blocking){
2. **int** flags = fcntl(fd, F\_GETFL);
3. **int** newflags = blocking ? (flags & ~O\_NONBLOCK) : (flags | O\_NONBLOCK);
4. **return** (flags == newflags) ? 0 : fcntl(fd, F\_SETFL, newflags);
5. }

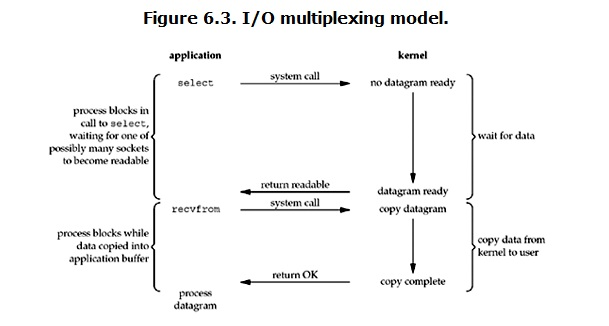
fcntl将recv设为非阻塞。

1. **int** fcntl(**int** fd, **int** cmd, ... /\* arg \*/ );

这里针对读写功能，设置为非阻塞后，和accept无关,accept要实现非阻塞需要用select()

IO multiplexing

也称event driven IO。改进了no-blocking（recvfrom过程相同），增加了专门的监控轮询function（select/poll/epoll）。



优势

poll/select一次可检测所有socket情况，单个process就可以同时处理多个网络连接的IO。由于经历两次system call，如果处理的连接数不是很高的话，不一定比multi-threading + blocking IO性能好。

IO multiplexing VS no-blocking

1.单个process就可以同时处理多个网络连接的IO。

2. process调用select后block到有socket中的数据准备好后recv，保证每次切换取读数据线程是有意义的。

select

select函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞。当select函数返回后，可以 通过遍历fdset，来找到就绪的描述符**。**

1. **int** select(**int** maxfdp,fd\_set \*readfds,fd\_set \*writefds,
2. fd\_set \*errorfds,**struct** timeval \*timeout);

maxfdp fd数量最大值

fd\_set：fd的集合，通过宏操作，FD\_ZERO(fd\_set \*);清空集合 。FD\_SET(int, fd\_set \*);将一个给定的文件描述符加入集合之中 FD\_CLR(int, fd\_set\*); 将一个给定的文件描述符从集合中删除。return正值：某些文件可读写或出错 负值select错误 0：等待超时，没有可读写或错误的文件

poll

1. **int** poll (**struct** pollfd \*fds, unsigned **int** nfds, **int** timeout);

和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符。不同在于select使用三个位图来表示三个fdset的方式，poll使用一个 pollfd的指针实现。

1. **struct** pollfd {
2. **int** fd; /\* file descriptor \*/
3. **short** events; /\* requested events to watch \*/
4. **short** revents; /\* returned events witnessed \*/
5. };

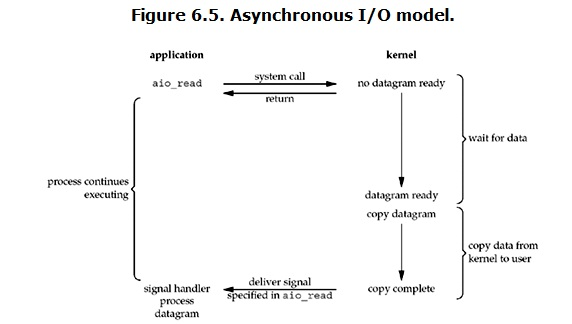
epoll

任一时间只有部分的socket是"活跃"的，select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。

epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

if linux kernel >= 2.6 is detected, then the java.nio.channels.spi.SelectorProvider will use epoll.

asynchronous IO



Socket Api

#include <sys/socket.h>

scoket🡪bind🡪listen🡪accept🡪recv🡪close

[mygithub](https://github.com/lovemooner/CMoon/blob/master/src/Thread100.cpp)

[Reference](https://blog.csdn.net/xc_tsao/article/details/44123331)

Unix下任何设备、管道、FIFO等都是文件形式，socket就是一个文件。

1. #include<sys/socket.h>
2. **int** socket(**int** domain, **int** type, **int** protocol);

Return 文件描述符fd,非负整数。即socket句柄，实际上是一个索引值

domain 协议族。AF\_INET,PF\_INET是IPv4

type 协议类型。SOCK\_STREAM/ SOCK\_DGRAM

listen

进入被动监听状态，不阻塞。

1. **int** listen(**int** sockfd, **int** backlog);

sockfd 套接字类型的文件描述符。

backlog请求队列最大长度。超过时客户端连接失败的错误或重传。

accept

接受客户端请求，建立TCP连接，完成三次握手。Request Queue为空时阻塞。

1. **int** accept(**int** listenfd, **struct** sockaddr \*addr, socklen\_t \*addrlen);

listenfd 服务端的fd

addr 客户端地址

return 返回客户端sock\_fd

IO优化

理想状态CPU，内存，IO外设都满负荷工作，大部分情况瓶颈都在IO的读写部分，NIO是优化了IO的读写。NIO最大化使用CPU资源。

**BIO VS NIO VS AIO**

* BIO：同步阻塞，程序直观简单，由于它的阻塞性, 每一个流的读写都要占用一个线程。适合连接少IO密集型。read/wrtite阻塞
* NIO：同步非阻塞。（连接数目多且每个连接只发送少量数据），比如一个聊天服务器。
* AIO：异步非阻塞。无需轮询。使用场景：连接数目多。比如相册服务器。

标准IO

标准IO是阻塞性, 每一个流的读写都需要占用一个线程. 这意味着,容器要为每个socket创建一个线程，但当线程的数量增加超过了 1K，标准 I/O 的不足就表现出来了。由于要在线程间进行上下文切换，因此 CPU 简直变成了超载。限制了server可处理的连接数，可伸缩性很差。

1. **protected** **void** doGet(HttpServletRequest req, HttpServletResp response)  {
2. ServletInputStream input = request.getInputStream();
3. **byte**[] b = **new** **byte**[1024];
4. **while** ((len = input.read(b)) != -1)
5. . . .

### 1.4.2 Java NIO

用一个线程管理大量网络连接，一次可以轮询所有网络连接的IO状态。避免创建大量的线程和线程切换造成资源的浪费，在高并发环境有较高吞吐量。

select 检测，>kernel 2.6 时epoll

socketChanell.read(buff) 数据从内核copy到用户空间。阻塞读，但一定是ready数据，只需数据copy时间。

异步处理

java nio提供了异步处理，这个异步应该是指编程模型上的异步。基于reactor模式的事件驱动，事件处理器的注册和处理器的执行是异步的。

非阻塞

调用select要block，select返回后再次对ready的文件描述符进行操作需要block。（）。

### 1.4.3 Java AIO

非阻塞：调用AIO函数会立即返回。由OS完成数据的拷贝，用户程序可继续执行。

异步：内核将I/O操作的数据copy到用户空间的缓存，并主动通知我们的程序。

异步 VS 同步

同步：用户线程发起I/O请求后等待或者轮询内核操作完成后才能继续执行

异步：由内核I/O通知线程或调用注册的回调函数

在使用新的异步IO时，主要有两种方式——Future轮询和Callback回调

## 1.6 JVM

**谈谈你对JVM的理解？**

Java的一个非常重要的特点就是与平台的无关性。而使用JVM是实现这一特点的关键。Java源文件经编译成字节码程序，通过JVM将每一条指令翻译成不同平台机器码，通过特定平台运行。

JVM执行程序过程 ：加载class文件 🡪管理并分配内存(生成对象) 🡪执行垃圾收集

new过程：JVM加载类🡺为对象分配内存🡪 调用构造函数初始化成员变量(<init>)

### 1.6.1 类加载机制

#### 类加载过程

Class对象自身的实例化过程。

注意：这里的加载指类的初始化过程，和类是否实例化无关。

[Oracle官方](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html#jvms-5.5)

[Reference](https://blog.csdn.net/ns_code/article/details/17881581)

类加载分了三个阶段，loading，linking和initializing。

1 )加载

1.加载Class编译后的二进制字节流到方法区

2.//在永久区中创建一个 instanceKlass**对象**。通过该对象访问方法区中的这些数据。

hotspot源码显示最后调用allocate\_permanent方法在PermGen分配内存，所以instanceKlass对象保存在永久代区域；

2 )链接

验证--准备--解析

**准备**

* 为类变量**分配内存**（方法区）空间并设置默认值。
* 不包括实例变量。实例变量会在对象实例化时随着对象一块分配在堆中。

**解析**

* JVM将常量池中的符号引用转化为直接引用。包括类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。
* 解析可能在initializing之前，也可能在之后。这是为了支持Java语言的运行时绑定（动态绑定）。

String s1=new String("abc")

符号引用：编译期s1； 直接引用：运行期s1转化成真实地址； 字面量: "abc"

3 )初始化

真正开始执行类中定义的代码**。**初始化阶段是执行类构造器<clinit>()方法的过程。

**<clinit>**

Code: ClassClinit100

编译器自动收集，类变量的赋值语句和静态语句块中的语句合并产生。收集顺序是由源文件中出现的顺序所决定的。如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生成此方法。



**<clinit> VS <init>**

两种都是编译器生成的方法。

**<clinit>：**是类（或接口）构造器，在类加载的初始化阶段对类进行初始化，类变量内存分配在方法区。

**<init>：**实例构造器。

<init>加入到构造器第一句执行



#### 触发条件

* JVM执行指令new/getstatic/putstatic/invokestatic/。
* 类反射。
* 当初始化一个类的时候，如果其父类还没有进行初始化，则需要先触发其父类的初始化。
* 当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先执行该主类。

具体Case分析

1.类的初始化后并未类的实例生成，比如getstatic….



2.单例中类Singleton初始化阶段会生成对象instance

这里生成对象instance和Singleton的初始化顺序？

Code: Singleton101



几个阶段是按顺序**开始**，而不一定按顺序**完成**，通常都是交叉进行的。这里<clinit>执行过程激活new的执行。

模块化

TODO4（jboss modules、osgi、jigsaw）

### 1.6.2 JVM内存管理

内存结构/运行时数据区



JVM区域总体分两类，Heap区和非Heap区。

#### Heap

所有类实例和数组对象分配内存的区。是各线程共享的运行时内存区域。包括Eden Space（伊甸园）&Survivor Space(幸存者区)&Old Gen（老年代）。



* Eden区：伊甸(yīdiàn)园。绝大多数刚创建的对象会被分配在Eden区（大对象可以直接被创建在老年代）。Eden是连续的内存空间，因此在其上分配内存极快。
* Survivor: Eden区分配的多数对象很快就会消亡，Eden区满的时候，执行Young GC，将消亡的对象清理掉，并将剩余的对象复制到空闲的Survivor。回收过程中，总是有一块 Survivor 区域是空闲的。
* Old区：YoungGC过程中,在两个Survivor区多次切换（HotSpot虚拟机默认15次）的长生命周期对象会将被复制到老年代。

-Xms：heap的最小内存 ，–Xmx：heap的最大内存

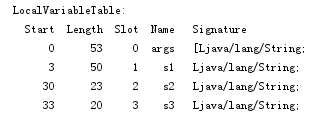
#### Stack

线程私有,与线程同时创建，又叫线程工作内存(高速缓存)，用于存储栈帧（Frames）。Frames随方法调用而创建，随方法结束而销毁。和方法一一对应。

|  |
| --- |
| 局部变量表 |
| 操作数栈 |
| 动态链接 |
| 方法返回地址 |
| 方法调用 |

局部变量表和操作数栈编译期确定，通过method\_info的code属性保存，供栈帧使用。

局部变量表：即字节码中LocalVariableTable，存放方法参数和方法内的局部变量。以变量槽（slot）为单位，slot为32位。



操作数栈：编译后中产生的操作数压入。

动态链接：指向运行时常量池中该栈帧所属方法的引用,持有这个引用是为了支持方法调用过程中的动态连接.

方法调用：调用方法的reference

程序计数器

寄存器，**保存当前执行指令的地址**，线程私有(每个线程都有自己的程序计数器)。

#### 方法区

很少GC，又叫永久区(Permanent)。用于存储类变量/final常量/编译的字节码/Class信息（field、方法、接口、版本等信息）等。

* -XX:PermSize=5M -XX:MaxPermSize=7M

由于PermGen经常引发OutOfMemoryError:PermGen，Java8彻底移除 PermGen，方法区移至Metaspace，字符串常量移至Heap。以便更灵活地被管理。

* -XX:MetaspaceSize 初始大小,设置的过大会延长垃圾回收时间。
* -XX:MaxMetaspaceSize 最大值，只受本地内存大小的限制，超过此值触发Full GC.

Todo

运行时常量池/本地方法栈

#### 堆外内存

off-heap memory, 又叫做直接内存（Direct Memory）,和堆内内存相对应。

堆外内存就是把内存对象分配在JVM以外的内存，这些内存直接受操作系统管理，在一定程度上减少GC对应用程序的影响。

DirectByteBuffer

将由C语言实现的native函数库分配在直接内存中，用存储在JVM堆中的DirectByteBuffer来引用。由于直接内存收到本机器内存的限制，所以也可能出现OutOfMemoryError的异常。

网络IO/文件的IO

Buffer地址地址传递给底层，有一个基本的要求，就是这个地址上的内容不能失效。而GC管理下的对象是会在Java堆中移动的。如果是非DirectBuffer，JDK会先创建一个DirectBuffer，再copy到堆中内存，再去执行真正的写操作。而使用DirectBuffer就会少一次内存拷贝，减少性能损耗。

Cache

cache的意义

缓解CPU和内存之间速度的不匹配问题。

原理

如果CPU需要的内容在cache里

局部性原理。/2-8理论

* 时间局部性：如果某个数据被访问，那么在不久的将来它很可能被再次访问；
* 空间局部性：如果某个数据被访问，那么与它相邻的数据很快也可能被访问；

cache的写回方式

cache的写操作方式可以追溯到大学教程《计算机组成原理》一书。

* write through（写通）：每次CPU修改了cache中的内容，立即更新到内存。每次CPU写共享数据，都会导致总线事务，导致总线事务的竞争，效率非常低。
* write back（写回）：每次CPU修改了cache中的数据，不会立即更新到内存，而是等到cache line在某一个必须或合适的时机才会更新到内存中；

无论是写通还是写回，在多线程环境下都需要处理缓存cache一致性问题。

线程安全

从cache的角度，产生线程不安全的原因：

当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。

如果一个变量在多个CPU中都存在缓存（一般在多线程编程时才会出现），那么就可能存在缓存不一致的问题

缓存一致性

1）通过在总线加LOCK#锁的方式

2）通过缓存一致性协议，比如MESI协议

MESI协议

处理缓存cache一致性问题。

### 1.6.3 GC

[Reference](http://www.cnblogs.com/ityouknow/p/5614961.html)

概括地说，GC根据一定的回收策略，自动的回收。保证JVM中的内存空间，防止出现内存泄露和溢出问题。

学习Java GC机制，可以帮助我们在日常工作中排查各种内存溢出或泄露问题，解决性能瓶颈，达到更高的并发量，写出更高效的程序。

#### 对象存活判定

目前虚拟机基本都是采用可达性算法。

**可达性算法**

GC Roots可达的对象是存活对象，不可到达的是GC回收的对象。GC Roots对象：

* 栈的局部变量表所引用的对象；
* 本地方法栈的JNI所引用的对象；
* 方法区的静态变量和常量所引用的对象；



1、2、4、6有GC Roots可达性，也就是存活对象，而实例3、5直接虽然连通，但并没有GC Roots与之相连，是GC需要回收的垃圾对象。

**引用计数器**

每个对象有一个引用计数器，当对象被引用一次则计数器加1，当对象引用失效一次则计数器减1，对于计数器为0的对象意味着是垃圾对象，可以被GC回收。

* 优点：1.实时回收 2.回收应用无需挂起3.回收具有区域性，不用扫描全部对象。
* 缺点 ：1.无法解决循环引用问题 2.浪费资源，即使内存够用，仍然在计数统计

4. 介绍GC 和GC Root不正常引用。

7. 数组多大放在 JVM 老年代（不只是设置 PretenureSizeThreshold ，问通常多大，没做过一问便知）

 老年代中数组的访问方式

11. 12. 如果想在 GC 中生存 1 次怎么办

深入分析了Classloader，双亲委派机制

#### 垃圾回收算法

直观的观察案例：[Reference](https://mp.weixin.qq.com/s?__biz=MzI4NDY5Mjc1Mg==&mid=2247486700&idx=2&sn=0e7081bb3016886fa45aa6952d8e2557&chksm=ebf6d493dc815d854d8262dcd20bde07befe475794efd3c7c3b254208fd6600e94c7690d70d6&scene=21#wechat_redirect)

特点：Java内存分配和回收的机制概括的说就是：分代分配，分代回收。

GC 分为Young GC(Minor GC）和Full GC。

**Young GC**

* GC触发条件**：** Eden区分配满的时候触发。
* 回收过程： 在Eden区创建对象 🡪Eden区处理消亡对象 🡪复制剩余对象到Survivor 🡪 清理出一块空闲的Survivor 🡪复制长周期对象到老年代

**Full GC**

* 触发条件： Metaspace /Old gen空间不足。从survivor复制到老年代时可用空间不足
* 回收范围：对整个新生代、老生代、元空间（metaspace）的全局范围的GC；。

如何减少GC时间和次数？

1.减少资源消耗 2.java ops不宜设置过大。

Q:jvm 如何分配直接内存， new对象如何不分配在堆而是栈上，常量池解析?

#### 垃圾收集器

[Reference](https://crowhawk.github.io/2017/08/15/jvm_3/)

**1）新生代收集器**

* Serial(串行)收集器:复制算法。最古老，最稳定以及效率高的收集器。单线程回收，GC过程中会Stop The World。
* ParNew:是Serial的多线程版本。回收过程也是暂停所有用户线程。
* Parallel Scavenge：复制算法。CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，适合与用户交互，而Parallel Scavenge收集器的目标是高效率地利用CPU时间，达到一个可控制的吞吐量（Throughput）。适合后台运算。

**2）老年的收集器**

* Serial Old： Mark-Compact算法。Serial的老年代版。单线程。
* CMS（Concurrent Mark Sweep）： Mark-Sweep算法。追求最短回收停顿时间为目标，重视响应速度。
* Parallel Old： Mark-Compact算法。Parallel Scavenge收集器的老年代版。

**3）G1垃圾收集器**

当今收集器技术发展最前沿的成果之一。

默认垃圾收集

jdk1.7: Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.8: Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.9: G1

-XX:+PrintCommandLineFlagsjvm参数可查看默认设置收集器类型

Mark-Sweep

标记-清除回收算法

1. 标记和清除过程的效率都不高 2.空间碎片太多。常用可达性算法。



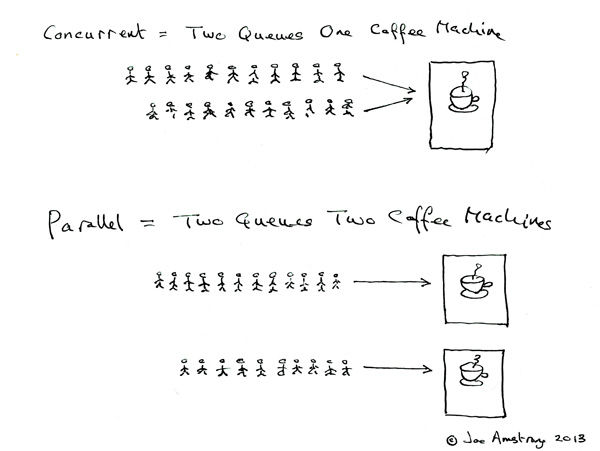
Copying

复制收集算法

这种算法的代价是将内存缩小为原来的一半，持续复制长生存期的对象则导致效率降低



Concurrent VS Parallel



#### GC优化

[**Reference**](http://www.ityouknow.com/jvm/2017/09/29/How-to-optimize-Java-GC.html)

JVM基本优化

* 设置-Xms，-Xmx参数
* 运行在server模式下（使用-server参数）
* 系统中没有残留超时日志之类的错误日志

进一步优化：

降低进入老年代的对象数量

老年代GC相对来说会比新生代GC更耗时（copy 算法）

* Jdk7+ 用G1收集器，
* NewRatio调节大小，case

减少Full GC的执行时间

Full GC比Minor GC要长很多，超过1s，可能出现超时错误。

* 减小老年代内存可减少Full GC时间，可能引起OutOfMemoryError或导致Full GC频率升高。
* 增加老年代内存来降低Full GC的频率，但GC的时间可能增加。

因此，你需要把老年代的大小设置成一个“合适”的值。

代码优化

使用StringBuilder或StringBuffer来代替String

Json序列化

### 1.6.4 JVM优化

#### 内存溢出&泄漏

**1）内存溢出**

申请了10个字节的空间，写入11或以上字节的数据，就会溢出。

java.lang.OutOfMemoryError，一般都是程序Bug或者JVM参数配置问题引起。

Java heap space

造成原因：吞吐量增加

解决方案

* 使用 -Xmx 增加堆大小
* 修复应用程序中的内存泄漏

PermGen space

* 应用中很多class，web服务器对JSP进行pre compile时
* webapp加载大量的第三方jar

解决方案

* 使用 -XX: MaxPermSize 增加 Permgen 大小
* 不重启应用部署应用程序可能会导致此问题。重启 JVM 解决

unable to create new native thread

造成原因：内存空间不足

解决方案

* 为机器分配更多的内存
* 减少 Java 堆空间
* 使用 -Xss 减小线程堆栈大小
* 增加操作系统级别的限制
* 修复应用程序中的线程泄漏。

其他

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247485517&idx=2&sn=112ca9b61ff3d5b6cc696f1f492eb809&chksm=96cd4801a1bac1176cc0b46bc0bef7f496adfac92e1d2ffd591c5e7e1eb3a9fd8fec44d2cf78&mpshare=1&scene=1&srcid=&pass_ticket=rMfcWI31qoBc%2BJleH2gnwJiWltH0srLHo3%2BPGnyWlwrmp8k7vmb0%2FxHQPasaYnLj#rd)

2）Memory leak

[Reference](http://blog.csdn.net/dongnan591172113/article/details/79044678)

对象申请的内存不再使用，因为仍被引用着，GC无法移除，导致内存泄漏。

内存泄漏可能导致内存溢出。

**1. 长生命周期的对象持有短生命周期对象的引用**

静态集合类的生命周期跟应用程序一样长，有可能会发生内存泄漏，看下面代码：



集合类如果是局部变量，在方法栈退出后就没有引用了会被jvm正常回收，不会造成内存泄露。而如果是全局性的变量提供这样的删除机制或者定期清除策略非常必要。

Java集合类已经提供了删除机制



**2.集合里的对象属性被改变**



**3.监听器**

在Java中，我们经常会使用到监听器，如对某个控件添加单击监听器addOnClickListener()，但往往释放对象的时候会忘记删除监听器

**4.各种连接**

Java中的连接包括数据库连接、网络连接和io连接，如果没有显式调用其close()方法，是不会自动关闭的，这些连接就不能被GC回收而导致内存泄漏。一般情况下，在try代码块里创建连接，在finally里释放连接，就能够避免此类内存泄漏。

**5.外部模块的引用**

调用外部模块的时候，方法传入对象，也应该注意防止内存泄漏。如：

public void register (Object A)

这个方法传入对象A模块的引用，这时需要B模块提供了去除引用的方法，如unregister()。这种情况容易忽略，而且发生了内存泄漏的话，比较难察觉，应该在编写代码过程中就应该注意此类问题。

**6.单例模式**

因为单例对象初始化后将在JVM的整个生命周期内存在，如果它持有一个外部对象（生命周期比较短）的引用，那么这个外部对象就不能被回收，而导致内存泄漏。如果这个外部对象还持有其它对象的引用，那么内存泄漏会更严重，因此需要特别注意此类情况。

StackOverflowError（堆栈溢出）

请求的栈的深度大于虚拟机所允许的深度

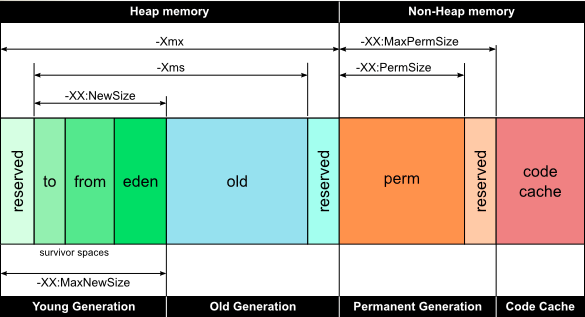
同步性

类加载由JVM自动同步，而执行对象的构造器可以中断的并发原子操作

#### JVM OPS

Heap

NewSize: young size



Heap

* -Xmn young的大小
* -XX:NewSize: young的大小
* –XX:MaxNewSize
* –XX:NewRatio: young:old,2表示1:2,即young/heap=1/5
* -XX:SurvivorRatio:2\*Survivor:eden,8代表1:1:8
* -server 启用JDK 的 server 版；一定要作为第一个参数，在多个CPU时性能佳

No Heap

* -XX:+UseParNewGC ：缩短minor收集的时间
* -XX:+UseConcMarkSweepGC ：缩短major收集的时间。此选项在Heap Size 比较大而且Major收集时间较长的情况下使用更合适

### 1.6.5 VM 指令集

[Reference](http://blog.csdn.net/dongnan591172113/article/details/51832690)

常量🡪栈顶。

|  |  |  |
| --- | --- | --- |
| iconst\_1 | 将int型的1推送至栈顶 | 4: iconst\_1 |
| fconst\_1 | 将float型的1推送至栈顶 |  |
| bipush | 将单字节(push byte)的常量值(-128~127)推送至栈顶 | 9: bipush 10 |
| sipush | 将一个短整型常量值(-32768~32767)推送至栈顶 |  |
| ldc | 超过32767。或者String常量 | 15: ldc #5 |

栈顶 ⬄ 本地变量

|  |  |  |
| --- | --- | --- |
| istore\_1 | 将栈顶int型数值存入第二个本地变量 |  |
| astore | 将栈顶**引用型数值**存入指定本地变量 |  |
| astore\_0 | 将栈顶引用型数值存入第一个本地变量 |  |
| astore | 将指定的引用类型本地变量推送至栈顶 |  |
| iload\_1 | 将第二个int型本地变量推送至栈顶 |  |
| aload\_0 | 将第一个引用类型本地变量推送至栈顶 |  |

pop

对栈顶进行操作

|  |  |  |
| --- | --- | --- |
| pop | 将栈顶数值弹出 |  |
| dup | 复制栈顶数值并将复制值压入栈顶 |  |

**invoke**

|  |  |  |
| --- | --- | --- |
| invokevirtual | 调用实例方法 |  |
| invokestatic | 调用静态方法 |  |
| invokeinterface | 调用接口方法 | Map.put |
| invokespecial | 调用超类构造方法/实例初始化方法/私有方法 |  |

5: invokespecial #3 // Method "<init>":(I)V

4: invokespecial #3 // Method "<init>":()V

域操作

成员变量

|  |  |  |
| --- | --- | --- |
| getstatic | 获取静态域，并将其值压入栈顶 |  |
| putstatic | 用栈顶的值为指定的类的静态域赋值 |  |
| getfiled | 获取实例域，并将其值压入栈顶 |  |
| putfileld | 用栈顶的值为指定实例域赋值 |  |

运算

|  |  |  |
| --- | --- | --- |
| iadd | 将栈顶两int型数值相加并将结果压入栈顶 |  |
| ishl | 将int型数值左移位指定位数并将结果压入栈顶 |  |

new及数组

|  |  |  |
| --- | --- | --- |
| new | 创建一个对象，并将其引用值压入栈顶 |  |
|  | 将int型数值左移位指定位数并将结果压入栈顶 |  |

javap

[Reference](http://blog.csdn.net/qq_24489717/article/details/53837493)

* -c 输出编译后的JVM指令



* -verbose 输出常量池，栈大小，方法参数。
* -l 输出行和变量的表。
* -public 只输出public方法和域
* -protected 只输出public和protected类和成员
* -package 只输出包，public和protected类和成员，这是默认的
* -p -private 输出所有类和成员
* -s 输出内部类型签名
* -constants 输出静态final常量
* 不加参数:只显示方法名
* -help 帮助

实例：

**int** a1=10;**int** a2=10;**int** a3=a1+a2;编译后的字节码



可见

变量赋值过程：数值通过命令的操作数push到栈顶->store到变量

Integer a1=10;



这里astore传的是引用！

局部变量。bipush/ldc等。

[Reference](https://blog.csdn.net/zhangjg_blog/article/details/22432599)

包含LineNumberTable和LocalVariableTable



LineNumberTable：描述了一条字节码和源码行号的对应关系

LocalVariableTable：局部变量与源代码中的局部变量之间的对应关系

### 1.6.6 Hotspot

JVM并没有根据Java对象直接通过VM映射到新建的C++对象，而是设计了一个oop/klass model。原因是不想让每个对象都包含vtbl(虚函数表)，减少虚函数表的消耗，所以就拆成klass和oop，其中oop中不含有任何虚函数，虚函数表保存于klass中，可以进行method dispatch。

为什么 C++ 中使用虚函数时会影响效率？

按照the C++PL的说法，虚函数调用比普通成员函数慢至多25%

oop/klass model

oop：Ordinary Object Pointer（普通对象指针），表示对象的实例信息，

klass：用来保存描述元数据和方法信息。oopDesc和klass区别

C++的多态实现原理

当涉及到多态性的时候，采用了虚函数和动态绑定，此时的调用就不会在编译时候确定而是在运行时确定。编译器会为每个有虚函数的类创建一个虚函数表。类的对象通过vptr指向虚函数表，可以遍历并调用其中函数。

## 1.8 算法结构

### 红黑树

红黑树是一种弱平衡二叉树。在相同的节点情况下，AVL树的高度低于红黑树。

### 查找算法

顺序查找：复杂度O(n)，缺点：的算法在数据量很大时显然是糟糕的

二分查找（binary search）：要求被检索数据有序。

二叉树查找（binary tree search）等。，只能应用于二叉查找树上

#### TreeMap

* 对key进行自然排序，
* 添加的对象需要实现Comparable接口（jdk自带的类型已实现）
* TreeMap内部基于红黑树排序

#### B+Tree

B+树非常适合文件系统

### Queue

队列，保证先进先出。

ConcurrentLinkedQueue

非阻塞

LinkedBlockingQueue

阻塞队列，对首和队尾有两把锁，ReentranLock

#### AQS

即AbstractQueuedSynchronizer 。 AQS实现FIFO等待队列，构建锁和同步容器提供了框架。Concurrent包内许多类都是基AQS构建。(区分上篇work的容器，这里是锁的容器。池容器用到的锁就是这里的锁)

CLH Lock

[Reference](https://www.jianshu.com/p/f43e581976b9)

CLH锁是一个自旋锁，提供先来先服务的公平性。线程必须等待占用的锁被释放后，通过竞争锁来获取资源的控制权。等待获取锁方式有“被动”和“主动”两种。

被动模式 VS 主动模式

被动模式：竞争线程自身进入阻塞状态，由某种调度机制唤醒；

主动模式：通常使用自旋的方式实现。竞争不激烈，锁资源很快就可以被释放的前提下，自旋锁的计算开销，要低于被动模式。

FIFO

在自旋模式下，依赖线程自身的抢占可能会导致其中某些线程始终无法获得锁，因此需要一个策略来保证争夺锁的线程可以以某种顺序获得锁的控制权，

实现方式：1.线程请求锁的顺序 2.线程优先级来。

AQS原理

AQS基于CLH锁的思想，主要从两方面进行了改造：**节点的结构与节点等待机制。**

* 在结构上引入了头结点和尾节点。
* 在等待机制上由原来的自旋改成阻塞唤醒。

实现细节

**AbstractOwnableSynchronizer**

1. **class** AbstractOwnableSynchronizer{
2. //当前占用该锁的线程
3. **private** **transient** Thread exclusiveOwnerThread;
4. //同步状态
5. **private** **volatile** **int** state;
6. **private** **transient** **volatile** Node head;
7. **private** **transient** **volatile** Node tail;
8. }

state

1 占用 0 未占用，需控制同步

Node

节点。

1. Node {
2. **int** waitStatus;
3. Node prev;
4. Node next;
5. Node nextWaiter;
6. Thread thread;
7. }

waitStatus

1.CANCELLED( 1)，表示当前节点的线程因为超时或中断被取消了；

2.（0） 初始化状态。表示当前节点在sync队列中，等待着获取锁。

3.SIGNAL(-1)，当前节点的后续节点中的线程通过 park 被阻塞了，需要通过unpark解除它的阻塞；

4.CONDITION(-2)，表示当前节点在等待condition，也就是在condition队列中；

5.PROPAGATE(-3)共享模式的头结点可能处于此状态，表示无条件往下传播，引入此状态是为了优化锁竞争，使队列中线程有序地一个一个唤醒

nextWaiter

节点模式。

FIFO链表

AQS构造的队列实际是Node的链表



AQS在结构上引入了头结点和尾节点，他们分别指向队列的头和尾，尝试获取锁、入队列、释放锁等实现都与头尾节点相关，并且每个节点都引入前驱节点和后后续节点的引用。

**ReentrantLock.lock()**

1. **final** **void** lock() {
2. acquire(1);
3. }

acquire

tryAcquire🡪 addWaiter🡪 acquireQueued

1. **public** **final** **void** acquire(**int** arg) {
2. **if** (!tryAcquire(arg) &&
3. acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
4. selfInterrupt();
5. }

selfInterrupt: Thread.interrupted()

tryAcquire

获取锁，根据state判断锁是否被占用。

fairTryAcquire

1. **protected** **final** **boolean** tryAcquire(**int** acquires) {
2. **final** Thread current = Thread.currentThread();
3. **int** c = getState();
4. **if** (c == 0) {
5. **if** (!hasQueuedPredecessors() &&
6. compareAndSetState(0, acquires)) {
7. setExclusiveOwnerThread(current);
8. **return** **true**;
9. }
10. }
11. **return** **false**;
12. }

Nonfair Lock没有hasQueuedPredecessors。这里乐观锁cas对state同步。

hasQueuedPredecessors

[Reference](https://blog.csdn.net/tomato__/article/details/25782747)

判断是否有等待线程 false没有，true有

1. **public** **final** **boolean** hasQueuedPredecessors() {
2. Node t = tail; // Read fields in reverse initialization order
3. Node h = head;
4. Node s;
5. **return** h != t &&
6. ((s = h.next) == **null** || s.thread != Thread.currentThread());
7. }
8. Queued没有元素
9. Queued有元素，但是currentThread Node

**addWaiter**

入队。通过CAS把当前线程追加到队尾

1. **private** Node addWaiter(Node mode) {
2. Node node = **new** Node(Thread.currentThread(), mode);
3. ... Node追加到队尾
4. }

acquireQueued

阻塞Node (after addWaiter)。

1. **final** **boolean** acquireQueued(**final** Node node, **int** arg) {
2. **boolean** interrupted = **false**;
3. **for** (;;) {  //自旋
4. **final** Node p = node.predecessor();
5. **if** (shouldParkAfterFailedAcquire(p, node) &&
6. parkAndCheckInterrupt())
7. interrupted = **true**;
8. }
10. }

**shouldParkAfterFailedAcquire**

阻塞前Set pred node waitStatus🡪SIGNAL ？

1. **private** **static** **boolean** shouldParkAfterFailedAcquire(Node pred, Node node) {         **int** ws = pred.waitStatus;
2. **if** (ws == Node.SIGNAL)
3. **return** **true**;
4. **if** (ws > 0) {
5. **do** {
6. node.prev = pred = pred.prev;
7. } **while** (pred.waitStatus > 0);
8. pred.next = node;
9. } **else** {
10. compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
11. }
12. **return** **false**;
13. }

**parkAndCheckInterrupt**

1. **private** **final** **boolean** parkAndCheckInterrupt() {
2. LockSupport.park(**this**);
3. **return** Thread.interrupted();
4. }

**unLock**

1. **public** **void** unlock() {
2. sync.release(1);
3. }

release

tryRelease🡪 unparkSuccessor 从head出释放等待队列

1. **public** **final** **boolean** release(**int** arg) {
2. **if** (tryRelease(arg)) {
3. Node h = head;
4. **if** (h != **null** && h.waitStatus != 0)
5. unparkSuccessor(h);  //
6. **return** **true**;
7. }
8. **return** **false**;
9. }

**tryRelease**

释放锁

1. **protected** **boolean** tryRelease(**int** releases) {
2. setExclusiveOwnerThread(**null**);
3. setState(0);
4. **return** **true**;
5. }

unparkSuccessor

 唤醒Node。release处传入head Node

1. **private** **void** unparkSuccessor(Node node) {
2. **int** ws = node.waitStatus;
3. **if** (ws < 0)
4. compareAndSetWaitStatus(node, ws, 0);
5. Node s = node.next;
6. **if** (s == **null** || s.waitStatus > 0) {
7. s = **null**;
8. **for** (Node t = tail; t != **null** && t != node; t = t.prev)
9. **if** (t.waitStatus <= 0)
10. s = t;
11. }
12. **if** (s != **null**)  //next node存在
13. LockSupport.unpark(s.thread);
14. }

## java8

#### Lambda

lambda就是匿名函数，用以简化匿名类。使代码简洁，可读性好。比如事件处理，集合处理，回调。可以传递函数。

函数式编程

面向对象编程是对数据进行抽象，以对象为中心，数据和行为组合成对象，不能独立存在，而函数式编程对行为进行抽象，以函数为中心。闭包让函数式编程语言有了对象的特性，而lambda让java有了函数式编程语言的特性。

闭包

闭包就是把函数以及变量包起来，使得变量的生存周期延长。闭包跟面向对象是一棵树上的两条枝，实现的功能是等价的。

闭包让函数式编程语言有了对象的特性，比如JavaScript(注意这里的对象不是面向对象，不附带OO的特性，仅指数据和行为的集合)。

语法

1. (Parameters) -> { Body }

Buildin接口

[Reference](https://www.w3cschool.cn/java/java-functional-interface.html)

位于包：java.util.function

#### Streams

一种集合运算方式，包括CRUD，聚合，统计，过滤等，原始iterator只能显示的遍历，stream采用内部迭代。像SQL查数据一样直观。让程序员写出高效率、干净、简洁的代码。

strings.stream().filter().map().limit().sorted().distinct()

这种风格将要处理的元素集合看作一种流，流在管道中传输，处理过程。

* 创建：Strings.stream()
* 中间操作：Pipelines,中间操作返回Streams本身。
* 最终操作：最终操作会消耗流，产生最终结果。forEach,count collect

parallelStream

当使用串行方式去遍历时，每个item 读完后再读下一个item。而使用并行去遍历时，数据会被分成多个段，其中每一个都在不同的线程中处理，然后将结果一起输出。Stream 的并行操作依赖Fork/Join 框架来拆分任务和加速处理过程。

适用场景

大数据量和多核

Spliterator

实现原理

[Reference](https://colobu.com/2014/11/18/Java-8-Stream/)

#### Optional

不用显示判空（很好解决空指针异常？）。

[Reference](https://www.jianshu.com/p/c169ddd34903)

orElse

返回默认值

1. **return** user.orElse((**null**);
2. //而不要
3. **if**(user.isPresent()) {
4. **return** user
5. }**else**{
6. **return**  **null**;
7. }

orElseGet

返回值由函数产生

1. **return** user.orElseGet(() -> fetchAUserFromDatabase());
2. //而不要
3. **if**(user.isPresent())
4. **return** user
5. **else**
6. **return**  fetchAUserFromDatabase();

ifPresent

1. user.ifPresent(System.out::println);
3. //而不要下边那样
4. **if** (user.isPresent()) {
5. System.out.println(user.get());
6. }

map

1. **return** user.map(u -> u.getOrders()).orElse(Collections.emptyList());
2. //上面避免了我们类似 Java 8 之前的做法
3. **if**(user.isPresent()) {
4. **return** user.get().getOrders();
5. } **else** {
6. **return** Collections.emptyList();
7. }

# 1.5 Thread

### 1.5.1 多线程

理想状态CPU，内存，IO外设都满负荷工作，多线程本质上是利用闲置的CUP资源，加快的处理速度。

使用场景

* 计算性Case。
* BIO：大部分情况瓶颈都在IO的读写部分，多线程处理更多请求，支持更改的并发。

多线程的代价

* 线程创建和销毁所花时间，以及系统资源开销(线程池)
* 线程调度, 上下文切换 ： 上下文切换过程并不廉价
* 内存同步的开销
* 设计更复杂

线程 VS 进程

进程是资源(CPU时间，内存等)分配的基本单位，线程是CPU调度的基本单位。

#### 线程状态

[Reference](http://blog.csdn.net/dongnan591172113/article/details/51967711)

New->Runable(start)->Running->Blocked

Blocked

Blocked VS Wait

* 线程未获得锁进去等待队列EntrySet（blocked）。已获锁主动释放，进入WaitSet（wait）。
* 从kernel角度，都是等待状态，没本质区别。
* jvm出于管理需要做了区分,Blocked由jvm唤醒，wait由另一个线程。

Interrupt

* 每个线程都有一个打断标志。线程在sleep/wait/join时，别的进程调用此线程的interrupt() 方法，此线程会被唤醒并被要求处理InterruptedException；
* 线程运行时执行interrupt，线程只会设置 “打断标志” 。

所以说，interrupt() 方法能够中断线程的等待过程,不会中断一个正在运行的线程。

#### Concurrent包

JDK5将Doug Lea的并发库引入到Java标准库中，提供新的启动(submit)、调度、管理线程的一大堆API了。增强了并发功能，简化了代码。主要特性：

* ReentrantLock
* 线程池
* 新增并发集合类ConcurrentHashMap，ConcurrentLinkedQueue等
* Semaphore

#### ThreadGroup

[Reference](http://www.cnblogs.com/0616--ataozhijia/p/3710571.html)

Code: ThreadGroup100

使用线程组的好处是可以对这一组的线程进行整体操作。

Thread.currentThread ().getThreadGroup ()

main方法默认group为main

#### Linux实现原理

[Reference](https://juejin.im/post/5a31d1be6fb9a04517053523)

linux用户态的进程、线程基本满足上述概念，但内核态不区分进程和线程。可以认为，内核中统一执行的是进程，但有些是“普通进程”（对应进程process），有些是“轻量级进程”（对应线程pthread或npthread），都使用task\_struct结构体保存保存。

使用fork创建进程，使用pthread\_create创建线程。两个系统调用最终都都调用了do\_dork，而do\_dork完成了task\_struct结构体的复制，并将新的进程加入内核调度。

TODO1:Linux线程实现

线程上下文切换

引起切换的原因

* **时间片**用完，CPU正常调度下一个任务
* 被其他优先级更高的任务抢占
* 执行任务碰到**IO阻塞**，调度器挂起当前任务，切换执行下一个任务
* **用户代码主动**挂起当前任务让出CPU时间
* 多任务抢占资源，由于没有抢到被挂起
* 硬件中断

#### 线程同步

多个线程访问共享变量，有线程安全问题，需要线程同步。同步指多线程环境，线程之间的协同和配合。线程同步通过加锁机制实现线程安全。

实现方法

同步：多个线程并发访问共享资源时，保证同一时刻只有一个（信号量可以多个）线程使用。

实现同步的方法：

* 临界区（CriticalSection）：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。进程内可用。
* 互斥量：**互斥量用于线程的互斥。只能为0/1。**一个互斥量只能用于**一个资源**的互斥访问，可跨进程使用。
* 信号量：**信号线用于线程的同步。可以为非负整数，**可实现**多个同类资源**的**多线程**互斥和同步。当信号量为单值信号量是，也可以完成一个资源的互斥访问。可跨进程使用。
* 事件：用来通知线程有一些事件已发生，从而启动后继任务的开始,可跨进程使用。
* synchronized的底层实现就用到了临界区和互斥锁（重量级锁的情况下）这两个概念。

java具体实现

* ThreadLocal
* sychronized
* Lock
* Atomic

线程安全

[Reference](http://www.jasongj.com/java/thread_safe/)

线程安全：多线程环境，如果每次运行结果和单线程运行的结果是一样的，就是线程安全的。

原因：Java多线程由于采用共享内存进行通信，线程安全问题都是由共享变量(全局变量及静态变量)引起的。

提到线程安全,需要关注原子性、可见性

****原子性****

[Reference](http://blog.psjay.com/posts/summary-of-java-concurrency-two-synchronized-and-atomicity/)

保证串行的顺序。原子操作：不可中断的一个或一系列操作。

An atomic operation is one that **cannot be interrupted** by the scheduler; if the operation begins, and then it will run to **completion** before the possibility of a context switch.

这里的原子性指java规范中定义的原子操作，指原始类型或引用类型的读取/赋值。

Synchronized/Lock

可实现更大范围操作的原子性。这里的原子性是互斥。可以中断，但即使中断了，组中的其他方法也不能被执行，保证是串行的顺序。

**在java多线程里指的原子性即为互斥，串行执行。**

原子性操作



只有语句1是原子性操作

x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。



x是共享变量，线程公有的。test方法读取变量到线程私有的栈中，x在内存中有两份copy，需要同步。和x的内存位置不同。

非原子操作：

**只要是非原子操作，操作就可能被中断或阻塞，需要同步来保证线程安全。**

****可见性****

[Reference](http://www.cnblogs.com/dolphin0520/p/3920373.html)

一些情况即便是原子操作也可能会引发错误，volatile保证可见性。

* 更新cache的修改到堆内存，并将其他CPU中对应的缓存行无效；
* 禁止指令重排。时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面。

使用恰当，比synchronized成本低(不会引起线程上下文的切换和调度)。

**Case 1**

1. **public** **class** Volatile101 **extends** Thread{
2. **private**  **boolean** isRunning = **true**;
3. **public** **void** run() { while (isRunning) {} }
4. **public** **void** doStop() { isRunning = **false**;}
5. **public** **static** void main(String[] args){
6. run();
7. doStop();
8. }
9. }

线程访问对象的值时，将堆内存中实例变量的值copy到工作内存。Java HotSpot Server版VM并不会将修改的值写回内存。

**Case 2 指令重排**

volatile还可以禁止指令重排，保证有序性



如果语句2会在语句1之前执行，线程2中就使用未初始化的context，导致程序出错

实现原理

修改volatile修饰的变量会立即更新到内存中，并将其它缓存中对该变量的缓存设置成无效，因此其它线程需要读取该值时必须从主内存中读取。这里更新内存的时间点具有不确定性只能用在上述场景，不能用于下面的自增操作



加入volatile关键字关键字时所生成的汇编代码（**非字节码**），会多出一个lock前缀指令

lock前缀指令，相当于一个内存屏障（也称内存栅栏）：

****有序性****

Java中可通过volatile在一定程序上保证顺序性，另外还可以通过synchronized和锁来保证顺序性。

### 1.5.5 线程锁

加锁是保护共享资源逻辑一致性的通用方案：在并发环境下，通过锁使得在某个时刻只有一个线程或进程可以访问共享的资源

#### 锁介绍

重量级锁

即悲观锁。假定有并发问题，读取数据时将数据锁定。需要OS参与，会引起线程阻塞和上下文切换。相对于乐观锁，系统资源开销大。

实现原理：monitor依赖OS的Mutex Lock来实现的。操作系统实现线程之间的切换需要从用户态转换到核心态，这个成本非常高，需要相对比较长的时间。

轻量级锁

轻量级锁即乐观锁。假定不会出现并发问题，每次不加锁，冲突失败就重试，直到成功为止。

原理：借助CPU提供的CAS（compare and swap）指令。适合在冲突不激烈的多线程环境。





ABA问题

潜在问题:如果需要关注A值变化过程，是会漏掉一段时间窗口的监控 .

CAS指令

CPU的CAS指令是原子性操作，执行完成前不可中断。本质上采用了缓存一致性协议（比如MESI），执行过程加了排他锁，不会出现线程安全问题。

注意：CAS操作只是实现了原子性,还需考虑可见性。

加锁优先级

无锁数据结构>锁区块>方法>对象锁>类锁

说明：尽可能使加锁的代码块工作量尽可能的小，避免在锁代码块中调用RPC方法。

##### 可重入锁

可重入锁，也叫做**递归锁**，指同一线程外层函数获得锁后，内层函数仍可获取该锁。

**特点：发生在同一线程内**

Code: ReentrantTest101



执行输出“method2”

可重入锁的最大作用就是可以避免死锁。这里讲的是广义上的可重入锁，而不是单指JAVA下的ReentrantLock。ReentrantLock 和synchronized 都是可重入锁

实现原理

state不为0，

##### 锁优化

JDK 1.6优化了并发锁，线程之间更高效地共享数据。

**锁粗化**:将多个连续的锁扩展成一个范围更大的锁，用以减少频繁互斥同步导致的性能损耗。

**锁消除:** JIT通过逃逸分析，去除堆上不会被其他线程访问到的代码锁。

**轻量级锁：**多个线程交替进入临界区，只要一条CAS原子指令就可以完成锁的获取及释放。

**偏向锁：**对轻量级锁进一步优化。使用CAS指令置换对象头ThreadID，下一次同一个线程进入则偏向该线程，无需任何同步操作。当有竞争，需解除偏向锁，进入轻量级锁

**适应性自旋:**为了避免线程频繁挂起、恢复的状态切换消耗。产生了忙循环（循环时间固定），即自旋。JDK1.6引入了自适应自旋。自旋时间根据之前锁自旋时间和线程状态，动态变化，用以期望能减少阻塞的时间。

##### 死锁

[Reference](https://juejin.im/post/5aaf6ee76fb9a028d3753534)

多进程在执行过程中，由于竞争资源或者通信而造成的一种阻塞的现象。

竞争的资源，可以是线程池里的线程、网络连接池的连接，数据库中数据引擎提供的锁，等等一切可以被称作竞争资源的东西。

如何排查？

执行jstack命令，查看线程是否一致处于WAITING状态。



#### Sychronized

实现线程的互斥串行操作。保证接口的原子性和可见性。

**实现原理**

在HotSpot虚拟机中，synchronized通过对象内部的监视器锁（monitor）来实现。

[Reference](http://www.cnblogs.com/dennyzhangdd/p/6734638.html)

修饰代码段：在JVM被编译为monitorenter、monitorexit指令来获取和释放互斥锁。

修饰方法：方法flags标记为ACC\_SYNCHRONIZED。

**Hotspot实现**

加锁值存在对象头部，即mark中。cas操作实现轻量级锁,但仅用于交替执行同步块的情况。对于同一Thread可用偏向锁，减少cas操作。多个线程在同一时刻进入临界区膨胀为重量级锁。依赖OS层面实现。

#### 1.5.5.2 ReentrantLock

实现互斥串行操作的

ReentrantLock VS synchronized

本质上synchronized的lock升级到重量级锁依赖OS的mutex lock，有线程切换，会引起线程阻塞和上下文切换。相对于乐观锁，系统资源开销大**（**required**）**。

* Synchronized的锁是JVM层面上实现的,是原生语法层面。ReentrantLock是API层面的互斥锁（用起来会复杂一些，需要写控制锁的代码，加锁和解锁都需要显式写出）。
* RLock更灵活、更强大，增加了锁轮训、超时(tryLock[time])、中断(lockInterruptibly)等功能。

#### 1.5.5.3 ReadWriteLock

Compare with synchronized

 synchronized的修饰区域每次只有一个线程可以访问。这种强烈的互斥性使得每次不管是读数据还是写数据都只能有一个线程可以操作。**在有多个读线程可以并行执行**的情况下，它并不是一个理想的选择。

Compare with volatile

使用读写锁的方式在单个写线程加多个读线程的情况下，其实差别不大。在使用volatile变量的时候，因为每次操作修改的结果对于全局都是可见的。那么在只有一个写线程的情况下，只有这个线程可以唯一修改数据。不会存在有几个写线程而产生的竞争条件。对于有**多个写线程的情况下，volatile变量就不能保证数据的一致性了**。而ReentrantReadWriteLock却可以有锁的机制保证互斥。它同时也尽可能保证了足够大的并行性。

#### 1.5.5.5 Semaphore

基于java同步器AQS

#### 1.5.5.6 Atomic

Atom包下的原子操作类实现了一种乐观锁，其本质是使用了CPU级别的CAS指令。

AtomicReference

对象引用变量读和写都是原子性的

原理：它是通过"volatile"和"Unsafe提供的CAS函数实现"原子操作

compareAndSet(expect,update)

LockSupport

比较底层，应用开发还是用Lock

park VS wait

[Reference](https://stackoverflow.com/questions/26534489/unsafe-park-vs-object-wait)

#### LimitLatch

#### CountDownLatch

countDown()和await()

### J.U.C

### 1.5.4 线程池

位于Concurent包。

作用

* 降低资源消耗：减少在创建和销毁线程上花的时间(线程复用)。
* 提高响应速度：任务到达时不需要等待线程创建就可以立即执行。
* 提高系统资源的管理性：线程池可以统一**管理、分配、调优和监控**，避免大量线程而导致消耗完内存或者“过度切换”的问题。

监控

如何检查线程池内线程运行情况，busy？ study监控API

**Thread busy**: jstack pid | grep " RUNNABLE" | wc –l

**Waitting Nums**: jstack pid | grep "WAITING" | wc –l

生产者-消费者

TODO

#### 四种线程池

1.newCachedThreadPool

根据处理需要，可灵活新建和回收线程。

缺点：有一定风险，若任务很慢或者阻塞且请求很多，就容易造成线程泛滥，会导致整个系统的假死（无法接收处理新的请求）。

使用场景：需要很极致的速度

WorkQueue：SynchronousQueue

执行顺序: 非公平的SynchronousQueue，无序

2.newFixedThreadPool

定长线程池，超出的线程会在队列中等待。可重新配置coreSize

1. **final** ExecutorService fixed = Executors.newFixedThreadPool(1);
2. ThreadPoolExecutor executor = (ThreadPoolExecutor) fixed;
3. executor.setCorePoolSize(4);

TODO

使用场景：

3.newSingleThreadExecutor

* 单线程化的线程池。同newFixedThreadPool的 workQueue都是LinkedBlockingQueue，保证所有任务按序(FIFO)执行。
* 同newFixedThreadPool(1)没有本质分别，目的在于防止重新配置(CoreSize)。

使用场景

1.一些不太重要的收尾，日志等工作可以放到单线程的线程中去执行。

1）.构造器

上面三种pool底层都直接生成ThreadPoolExecutor。而SinglePool生成FDeExecutorService，本质上还是封装了FixedThread(ThreadPoolExecutor,core:1,max:1)。**public** **static** ExecutorService newSingleThreadExecutor() {

1. **return** **new** FinalizableDelegatedExecutorService
2. (**new** ThreadPoolExecutor(1, 1,0L, TimeUnit.MILLISECONDS,**new** LinkedBlockingQueue<Runnable>()));

2）. newSingle不能监控

继承自ExecutorService，所以无法Cast成ThreadPoolExecutor做监控。其实也不需要监控

4.newScheduledThreadPool

定长线程池，支持定时及周期性任务执行。

#### ThreadPoolExecutor

1. ThreadPoolExecutor **extends** ExecutorService **implements** Executor{
2. **private** HashSet<Worker> workers ;
3. **privite** BlockingQueue<Runnable> workQueue
4. **public** ThreadPoolExecutor(**int** corePoolSize,**int** maximumPoolSize,
5. **long** keepAliveTime,TimeUnit unit,
6. BlockingQueue<Runnable> workQueue) { }
7. }

* corePoolSize: worker数。
* maximumPoolSize**:**

**1.**newFixedThreadPool的corePoolSize和maximumPoolSize的大小是一样的

2. 参数对无界queue无意义。比如LinkedBlockingQueue/ SynchronousQueue。

* BlockingQueue超过corePoolSize，用来存储等待执行的任务。
* RejectedExecutionHandler ：线程拒绝策略。RejectPolicy任务可以拒绝：目前集中拒绝策略均可。任务不可决绝：CallerRunsPolicy可以返回到main中执行
* KeepAliveTime超过corePoolSize数的空闲线程存活时间
* RejectedExecutionHandler：blockQueue满了后对新加入线程处理策略

#### WorkQueue

1. **public** **interface** BlockingQueue{
2. //添加
3. add(anObject):添加成功返回**true**,否则抛异常
4. offer(anObject):添加成功返回**true**,否则返回**false**.
5. put(anObject): 添加时没有空间,调用的线程阻断到有空间再继续.
6. //移除
7. poll(time):取走queue排在首位的对象,time时间内一直循环，取不到时返回**null**
8. take():取走queue里排在首位的对象,若queue为空, 阻塞到有新的对象加入。
9. }

实现类

1）LinkedBlockingQueue

排序：以FIFO 。

大小：默认由Integer.MAX\_VALUE来决定.

2）SynchronousQueue

[Reference](https://zhuanlan.zhihu.com/p/29227508)

无界队列，缓存值为1的阻塞队列。对其的操作必须是放和取交替完成的。

CachedPool基于它实现，若有线程在执行，插入的线程（比如main）一直阻塞。

**公平模式 VS 非公平模式**

**公平模式**：会采用公平锁，并配合一个FIFO的queue来阻塞多余的生产者和消费者，从而体系整体的公平策略；

**非公平模式**（默认）：采用非公平锁，同时配合一个LIFO的stack管理多余的生产者和消费者，如果生产者和消费者的处理速度有差距，则很容易出现饥渴的情况，即可能有生产者或消费者的数据永远都得不到处理。

3）PriorityBlockingQueue

类似于LinkedBlockQueue,但其所含对象的排序不是FIFO,而是依据对象的自然排序顺序或者是构造函数的Comparator决定的顺序.

4）ArrayBlockingQueue

规定大小的BlockingQueue,其构造函数必须带一个int参数来指明其大小.其所含的对象是以FIFO顺序排序的。

使用方式：

1. ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

Executors

1. **public** **static** ExecutorService newFixedThreadPool(**int** nThreads) {
2. **return** **new** ThreadPoolExecutor(nThreads, nThreads,
3. 0L, TimeUnit.MILLISECONDS,
4. **new** LinkedBlockingQueue<Runnable>());

maximumPoolSize = Integer.MAX\_VALUE(Integer是4位，-2^31~2^31-1)

pool内线程顺序

**1)FixedPool/SinglePool**

workers：ThreadPoolExecutor在addWorker时加了锁，所以workers的执行顺序为添加顺序即程序调用顺序。阻塞时workQueue：LinkedBlockingQueue同workers也是添加顺序。

**2)CachedThreadPool**

SynchronousQueue的非公平模式生产者或消费者的数据永远都得不到处理。

#### 实现原理

ThreadPoolExecutor

1. **public** **class** ThreadPoolExecutor{
2. **private** **volatile** **int** corePoolSize;
3. **private** **volatile** **long** keepAliveTime;
4. **private** **volatile** RejectedExecutionHandler handler;
5. **private** **final** HashSet<Worker> workers = **new** HashSet<Worker>();
6. **private** **final** BlockingQueue<Runnable> workQueue;
7. }

* workers
* RejectedExecutionHandler：blockQueue满了后对新加入线程处理策略

addWorker

[Reference](https://blog.csdn.net/c10WTiybQ1Ye3/article/details/80428945)

addworker🡪ThreadPoolExecutor.run()🡪runWorker🡪 getTask() 🡪task.run()

* 从runWorker后currentThread已变成池中worker线程。

调用pool的execute时执行

1. **private** **boolean** addWorker(Runnable firstTask, **boolean** core){
2. ReentrantLock mainLock = **this**.mainLock;
3. mainLock.lock();
4. Worker w = **new** Worker(firstTask);
5. workers.add(w);
6. w.thread.start(); //开始执行
7. mainLock.unlock();
8. }

线程被包装成worker🡪添加后执行w.thread.start()🡪Worker中thread就是ThreadPoolExecutor本身，start run()🡪runWorker

Worker

1. **private** **class** Worker **extends** AbstractQueuedSynchronizer **implements** Runnable{   **final** Thread thread;
2. Runnable firstTask;  //添加的线程
3. **public** **void** run() {
4. runWorker(**this**);
5. }
6. }

runWorker

1. **final** **void** runWorker(Worker w) {
2. Runnable task = w.firstTask;
3. **while** (task != **null** || (task = getTask()) != **null**) {
4. task.run();
5. }
6. }

currentThread: ThreadExecutor本身

cachedPool：addWorker(null,**false**),firstTask为null

getTask

1. **private** Runnable getTask() {
2. **boolean** timedOut = **false**;
3. **for** (;;) {
4. **boolean** timed = allowCoreThreadTimeOut || wc > corePoolSize;
5. Runnable r = timed ?
6. workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
7. workQueue.take();
8. **if** (r != **null**)
9. **return** r;
10. timedOut = **true**;
11. }
12. }

Execute

线程池原理即Execute执行逻辑

1. **public** **void** execute(Runnable command){
2. **if** workCount<coreSize  //1
3. addWorker（**thread**,**true**）  //2
4. **else** **if**(workQueue.offer(command))//3
5. if workerCount==0 { addWorker(null,**false**))};  //4
6. **else** (!addWorker(command,**false**))
7. reject(command);  //5
8. }

**CachedPool**

coreSize=0

1.第一个thread进来会执行到4（coreSize为0会跳过1）

2.addWorker最终调用getTask()在keepAliveTime一直轮询到有task加入。

3.后面的线程走步骤 3（先进入workQueue，workCount此时为1，不会执行步骤4）。从而在1中的getTask在keep时间里能复用。

4.超过poll的time，workcount=0, 重复1,2,3

5: 若3中未添加成功会执行5 reject

**FixedPool**

keepAliveTime=0,workQueue: LinkedBlockingQueue

1. 执行1 创建核心线程
2. getTask若没任务会一直阻塞在blockQueue.task()
3. 可见超过coresize进入blockQueue的才会复用线程。

拒绝策略

blockQueue满了后对新加入线程处理策略

RejectedExecutionHandler

ThreadPoolExecutor.AbortPolicy() 抛异常

ThreadPoolExecutor.DiscardPolicy 丢弃被拒绝的任务。

ThreadPoolExecutor.DiscardOldestPolicy 丢弃队列中最老的任务

ThreadPoolExecutor.CallerRunsPolicy 直接在 execute 方法的调用线程中运行被拒绝的任务；如果执行程序已关闭，则会丢弃该任务。

池状态

区别于线程的状态。

1. **volatile** **int** runState; //当前状态
2. **static** **final** **int** RUNNING    = 0;
3. **static** **final** **int** SHUTDOWN   = 1;
4. **static** **final** **int** STOP       = 2;
5. **static** **final** **int** TERMINATED = 3;

SHUTDOWN: 调用shutdown()，pool不接受新的任务并等待所有任务执行完毕；

STOP: 调用shutdownNow()，pool不接受新的任务并尝试终止正在执行的任务；

TERMINATED: 处于SHUTDOWN或STOP之后状态，所有workers/workQueue已经销毁。

ctl

1. **private** **final** AtomicInteger ctl = **new** AtomicInteger(ctlOf(RUNNING, 0));

ctl包含两个field，workerCount和runState，workerCount是占据着一个atomic integer的后29位的，而状态占据了前3位，所以，workerCount上限是(2^29)-1。

workerCount

表示当前有效的线程数，也就是Worker的数量

**DB Pool源码分析**

[Reference](https://blog.csdn.net/shuaihj/article/details/14223015)

TODO4

### 1.5.6 ThreadLocal

实现原理

ThreadLocal为每个线程提供一份共享变量的的副本。

1. **public** **void** set(T value) {
2. Thread t = Thread.currentThread();
3. ThreadLocalMap map = getMap(t);
4. map.set(**this**, value);
5. }

* ThreadLocalMap为Thread的成员变量
* map的可以为ThreadLocal对象
* 变量在线程间隔离，不是在线程间共享，因此不存在线程安全和锁的问题

使用场景

变量在线程间隔离而在方法和类间共享的场景。

实例：数据库连接、Session管理等。Spring中使用

1. **private** **static** **final** ThreadLocal threadSession = **new** ThreadLocal(); //静态
2. **public** **static** Session getSession() **throws** InfrastructureException {
3. Session s = getSessionFactory().openSession();
4. threadSession.set(s);
5. **return** s;
6. }

ps: 该场景ThreadLocal使得实现更简洁。不用传递参数

内存泄漏

[Reference](https://mp.weixin.qq.com/s?__biz=MzI3NzE0NjcwMg==&mid=2650122904&idx=2&sn=0b4932987d63c8e6bfd01c4940882857&chksm=f36bb7b9c41c3eafad4374d70df53d4f1a0702a667bf5d65a8e045f3a752610acbe4d5a0de40&mpshare=1&scene=1&srcid=#rd])

### 1.5.7 Future模式

异步调用

调用立即返回，可处理其他事情，用于耗时操作，异步计算。CountdownLatch也可以处理类似场景,但 Future可获取返回值或Exception。

1. ExecutorService executor = Executors.newSingleThreadExecutor();
2. Future<String> future = executor.submit(**new** Callable<String>() {...});
3. String result = future.get();

API

isDone()： 查询执行状态

get(): 阻塞获取执行结果。 LockSupport.park(this)

**CompletableFuture**

[Reference](https://juejin.im/post/5b4622df5188251ac9766f47)

处理复杂任务。比如多个异步计算结果合并, thenApply, thenAccept, thenCompose

1. future = CompletableFuture.supplyAsync(() -> {**return** 10; });
2. future.thenAccept(System.out::println);

### 1.5.8 fork-join

[Reference](https://www.liaoxuefeng.com/article/001493522711597674607c7f4f346628a76145477e2ff82000)

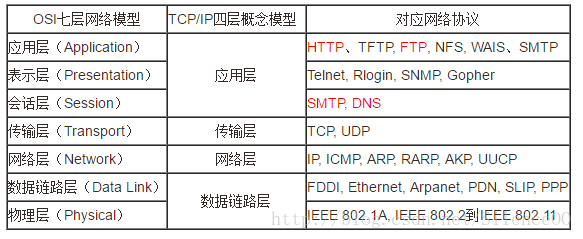
TODO3

ForkJoinPool

# 第二篇 Java Web

## 网络技术

### 2.1.1 协议分层



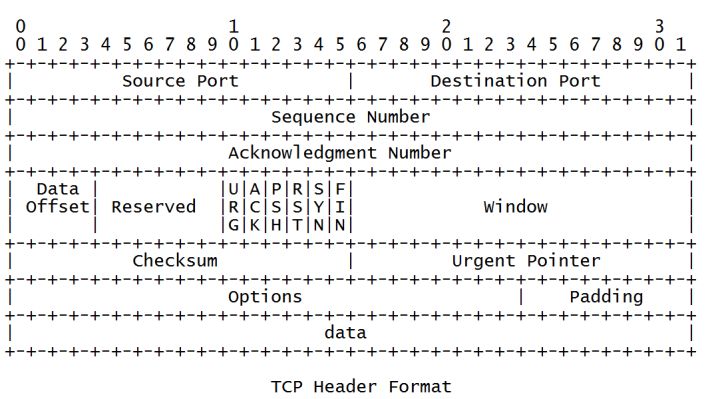
### 2.1.2 TCP

[Reference](https://github.com/jawil/blog/issues/14)

如何保证传输可靠性？**M**

TCP vs UDP

#### TCP头部



一行32位 4byte，options和padding可选，所以header至少20bytes

TODO: 通过浏览器看到的request header存在哪个字段？

Header

Sequence Number 标识发送数据

Acknowledgment Number：接收端确认数，并标识期望序号

Window

Options

Data

应用层数据包，比如HTTP

TODO 协议大小 [Refer](http://www.ruanyifeng.com/blog/2017/06/tcp-protocol.html)

#### 三次握手



#### 四次挥手



客户端收到FIN=1才Close,

**State**

LISTEN 服务监听

ESTABLISHED 意思是建立连接。表示两台机器正在通信

CLOSE\_WAIT 对方主动关闭连接或者网络异常导致连接中断

TIME\_WAIT: 我方主动调用close()断开连接，收到对方确认后状态变为TIME\_WAIT

**TIME\_WAIT占用的资源：**少量内存（查资料大概4K）和一个fd。

#### TIME\_WAIT

产生原因

解决方案

TODO3:三次握手/窗口滑动机制

### 2.1.3 HTTP

[Reference](https://developer.mozilla.org/zh-CN/docs/Web/HTTP)

超文本传输​​协议，用于传输诸如HTML的**超媒体文档**的应用层协议。它被设计**用于Web浏览器和Web服务器之间的通信**。基于TCP/IP层（也可以在任何传输层上使用,如UDP）。

1991 HTTP/0.9

服务器只能回应HTML，发送完即关闭TCP连接。

1996 HTTP/1.0

无连接

背景：早期浏览器与服务器交互数据频率低，发送数据完毕，连接就关闭，及时释放资源。

缺点：TCP连接的新建成本很高，因为需要客户端和服务器三次握手，性能比较差。

无状态

http的无状态是指每个请求独立的与上下文无关。

背景：早期web多用于浏览，服务器完成响应即关闭连接，无状态解放了服务器。

解决方案 Cookie/Session，缺点在于每次请求会传输大量重复的内容信息。

1997 HTTP/1.1

* 长连接：不关闭连接。通过Connection: keep-alive/close控制，默认开启。对于同一个域名，大多数浏览器允许同时建立6个持久连接。实现原理:
* 管道机制（pipelining）：客户端可以同时发送多个请求，服务器还是按顺回应
* Content-Length：服务器告诉浏览器完成回应，根据此字段判断response完成。在1.0版中，浏览器发现服务器关闭了TCP连接，就表明收到的数据包已经全了。
* 分块传输编码：对于一些很耗时的动态操作来说，这意味着，服务器要等到所有操作完成，才能发送数据，显然这样的效率不高。更好的处理方法是，产生一块数据，就发送一块，采用"流模式"（stream）取代"缓存模式"（buffer）

2015 HTTP/2

二进制协议：http/2是彻底的binary（http/1.1之前head必须是文本body可以是binary）。

多工：双向、实时通信。

数据流：

Method

GET/POST

1.根据HTTP规范，GET用于信息获取，而且应该是安全的和幂等的。POST表示可能修改变服务器上的资源的请求

2.GET请求数据一般会附在URL后,如：xx?n=1。POST在是HTTP包的body中。

3.GET方式提交的数据最多只能是1024字节，理论上POST没有限制，可传较大量的数据

Http 3

### 2.1.4 HTTPS

http的信息传输完全以明文方式，不做任何加密。https防止通信过程中被第三方获取明文。1994 网景创建，基于SSL。

证书颁发机构（CA）

中间人截获了公钥Key1后，可以偷天换日，自己另外生成一对公钥私钥，把自己的公钥Key3发送给接受者

## 2.2 J2EE

### 2.2.1 Servlet规范

J2EE是什么？

J2EE是由SUN提出的用于简化开发企业级应用程序的一系列规范的组合，简化企业应用开发，管理和部署，包含内容：

Servlet /JDBC/JNDI/JSP/EJB/XML/JMS/RMI/ JTA /JavaMail/JAF

核心是EJB，现在spring是事实的标准。

Servlet接口

Servlet本质就是一个接口，是一套处理网络请求的规范，主要为了扩展Java的web功能。

应用开发人员：根据规范编写业务程序。

Servlet容器：处理HTTP请求，封装request给server，response给客户端。

版本发展

* Servlet2.5 onwards：… 🡪tomcat6
* Servlet3.0：Aync Servlet;新增注解Filter,Listener；🡪Tomcat7
* Servlet3.1： Non-blocking I/O；升级HTTP协议成WebSockets; 🡪Tomcat8
* Servlet4.0：HTTP/2

#### API简介

1. **public** **interface** Servlet {
2. **void** init(ServletConfig var1);
3. ServletConfig getServletConfig();
4. **void** service(ServletRequest var1, ServletResponse var2)
5. String getServletInfo();
6. **void** destroy();
7. }

init()和destroy() 管理生命周期，还有一个处理请求的service()

Security

自定义filter

Spring-security

Shiro

权限

Oauth

生命周期

实例化🡺init🡺service🡺destroy。

判断未登录或session过期

1. request.getSession (**false**) ==**null**
2. request.getSession (**false**).getAttribute (“user”) ==**null**

Filter

过滤request。执行顺序跟xml文件中定义的先后关系有关。过滤字符编码，逻辑判断，如是否登录，权限控制

Filter vs AOP

AOP无法拦截直接访问静态资源的请求。Filter仅仅过滤请求。

Filter主要是针对URL地址做一个编码的事情、过滤掉没用的参数、安全校验（比较泛的，比如登录不登录之类），太细的话，还是建议用interceptor。

Listener

随web应用启动而启动，只初始化一次，随web应用的停止而销毁。主要用做一些初始化

#### Non-blocking NIO

[Reference](https://docs.oracle.com/javaee/7/tutorial/servlets013.htm)

Servlet 3.0对请求的处理虽然是异步的，但IO操作依然是阻塞的。Servlet 3.1 加入了非阻塞 I/O 并且引入了两个新的接口 ReadListener 和 WriteListener。

1. inputStream.setReadListener(**new** ReadListener() {
2. **void** onAllDataRead()   {
3. executor.execute(() -> {
4. **new** LongRunningProcess().run();
5. ctx.getResponse().getWriter().write("Hello World!");
6. ctx.complete();
7. });

Read http body（Write response）是阻塞，non-blocking即使释放线程。

Aync Servlet

Servlet3中新增AsyncContext

1. AsyncContext context = request.startAsync();
2. ctx.start(**new** Thread(ctx));  //向容器申请线程

对性能的改进不大，相当于闲置下了一个线程，但同时又占用了另一个线程。

### 2.2.2 对象的序列化

对象存在于JVM，序列化是一种对象持久化的手段，使对象可以被表示为一个字节序列（该字节序列包括该对象的数据、对象的信息和存储在对象中数据的类型），便于传输。

1）RPC（远程方法调用）

实现Java的Serializable接口，在两个系统之间的DTO对象里面可能会用到，

2）数据传输

xml和json两种数据格式用得更多一些。Json

1. // 将Java对象序列化为Json字符串
2. String objectToJson = JSON.toJSONString(initUser());
3. // 将Json字符串反序列化为Java对象
4. User user = JSON.parseObject(objectToJson, User.**class**);

## 2.3 Spring

Spring几乎涉及Java企业级开发的所有方面，已成Java Web开发事实上的标准（框架）。

* IoC和Aop 让软件内部做到组件(类)之间的低耦合，易替换，易扩展。
* 降低应用程序开发的复杂度。提供了Spring Core, Security, JPA等。
* 它可以集成其他框架，如 Structs、Hibernate、EJB 等。

发展

第一阶段：Core、Security、Data 简化单体应用开发，提供数据库访问、MVC等必要功能。

第二个阶段: Spring Boot 简化配置，提升开发效率。

第三个阶段：Spring Cloud 推动了微服务架构的落地。

第四个阶段 Spring Cloud Dataflow+容器

Dataflow的思想是不管是做实时消息处理的服务还是临时运行的任务，都可以认为是服务的组件，如果可以有一套DSL来定义这些组件之间的交互方式，然后在容器中进行自由组合、部署、伸缩，那么架构会非常灵活。

### 2.3.1 IoC

* IoC **：**Inversion of control 控制反转。反转指上层模块对于底层模块的控制交给IoC容器。控制指底层模块的实例化。IoC容器负责对象创建，高层模块直接使用
* 好处：依赖于抽象是面向对象设计的精髓，IoC对依赖关系进一步的解耦。本质都是为了代码更加的“高内聚,低耦合”。

IoC容器可理解为手机厂商，不同在于零件要自己生产，好处在于不用做组装工作。

IoC VS 工厂模式

Spring IoC本质上是工厂模式与反射机制的综合应用。

实现原理：

TODO: BeanFactory [Reference](https://mp.weixin.qq.com/s/yhoFcEk6Cxmzl0QxNSK4UA)

Dependency injection（依赖注入，简称DI），Ioc具体实现的其中一种。

* 构造函数中注入 ：后期无法更改。
* setter 方式注入 ：使用时要判空，没有构造器中new有保证。
* 接口注入

### 2.3.2 AOP

方便进行面向切面的编程，很多传统OOP不容易实现的功能可以通过AOP轻松应对。

自定义AOP

Aspectj静态代理

1. @Aspect
2. **public** **class** LogInterceptor{
3. @Pointcut("execution(\* love.moon.spring.controller.\*.\*(..))")
4. **private** **void** businessService() {}
5. @Around("businessService()")
6. **public** Object doAroundTask(ProceedingJoinPoint pjp) {}
7. }

另外还有注解 @AfterReturning，@AfterThrowing

Xml配置



pointcut切面

CGLIB动态代理

如果目标对象有实现接口，使用 JDK 代理。如果目标对象没有实现接口，则使用 CGLIB 代理

JDK动态代理

动态代理类在程序运行期间通过反射等机制动态生成。JDK针对接口

### 2.3.3 Spring MVC

工作原理

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247485021&idx=1&sn=acc1e9694239ca855ca8a7e077a467ad&chksm=9bed272fac9aae39038a35b5ad4e25b9e3471effa65c346e96f06bac9ddb0ac99b8ce909f941&mpshare=1&scene=1&srcid=#rd])

基于Servlet。

Request🡪 DispatcherServlet (web.xml)🡪 HandlerMapping(All Controller Map)🡪 Controller

在HandlerMapping中找到对应服务方法并调用。这里是反射call，CglibAopProxy静态代理。

[手写Spring MVC框架](https://mp.weixin.qq.com/s?__biz=MzI2OTQ4OTQ1NQ==&mid=2247485748&idx=1&sn=764704b0973c9bae26fb963fd50ba295&chksm=eadecd74dda9446263abfd5a5cbf21ee6988afda963bc5778b987cb5cf28c8ea34b698dde6ab&mpshare=1&scene=1&srcid=#rd])

TODO： Spring ModelAndView

MVC分层好处

1. 有利于分工
2. 复用
3. 好扩展，好维护

### 2.3.4 事务管理

事务是指一组操作完成一个任务，保证数据的完整性。Spring对事务进行封装，其最终还是在数据库层面实现的。

#### 声明式事务

Sping支持编程式事务和声明式事务，分离业务和事务管理代码。

applicationContext.xml配置

1. **<tx:advice** id="txAdvice" transaction-manager="transactionManager"**>**
2. **<tx:attributes>**
3. **<tx:method** name="get\*" propagation="REQUIRED" read-only="true"**/>**
4. **<tx:method** name="save\*" propagation="REQUIRED" isolation="DEFAULT"**/>**
5. **</tx:attributes>** **</tx:advice>**
6. **<aop:config>**
7. **<aop:pointcut** id="bds" expression="execution(\* love.service.\*.\*(..))"**/>**
8. **<aop:advisor** advice-ref="txAdvice" pointcut-ref="bds"**/>**
9. **</aop:config>**

Propagation传播行为，ServiceB.methodB 🡪 ServiceA.methodA

* REQUIRED：没有就起一个新的事务。methodA或methodB出现异常，事务回滚。
* SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行。
* 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。
* REQUIRES\_NEW：新建事务，如果当前存在事务，把当前事务挂起。
* MANDATORY：支持当前事务，如果当前没有事务，就抛出异常。
* NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

2) Isolation与数据库的隔离级别相对应。

3) Rollback-for人为控制事务

4) Read-only 保证可重复读。mysql默认就是Repeatable-Read。 Oracle默认是read commit，设置readonly后能达到同样效果。

Annotation

@Transactional 注解

类级别：表示所有该类方法都配置相同的事务属性信息。

方法级别：方法事务属性信息会覆盖类级别的相关配置信息。

只读事务

[Reference](https://www.cnblogs.com/hackem/p/3890656.html)

设置了readonly后，connection都会被赋予readonly，效果取决于数据库的实现



Mysql：true的时候会读到最新值

#### 实现原理

以管理Hibernate Session为例，基于AOP，

1. **public** **class** TransactionInterceptor {
2. **private** **static** **final** ThreadLocal threadSession = **new** ThreadLocal();
3. **void** aopProxyExecute(){
4. Session s = getSessionFactory().openSession();
5. threadSession.set(s); //绑定后，在dao层直接调用getCurrentSession()即可
6. session.beginTansaction();
7. service.callMethod();  //业务逻辑，不用写事务管理代码。提高开发效率。
8. session.commit();
9. session.close();  // 这里关闭，Hibernate的会话不自动关闭
10. }
11. }

JdbcTemplate事务

Mybatic事务

### 2.3.9 Spring Data JPA

[Reference](http://www.ityouknow.com/springboot/2016/08/20/spring-boo-jpa.html)

基于Hibernate的Jpa框架。

1. **public** **interface** UserRepository **extends** JpaRepository<User, Long> {
2. User findByUserName(String userName);
3. }

1）继承JpaRepository接口，拥有了基本CRUD功能以及分页功能

1. @Test
2. **public** **void** testBaseQuery() **throws** Exception {
3. User user=**new** User();
4. userRepository.findAll();
5. userRepository.findOne(1l);
6. userRepository.save(user);
7. userRepository.delete(user);
8. userRepository.count();
9. userRepository.exists(1l);
10. // ...
11. }

2）自定义简单查询，根据方法名自动生成SQL

findByXX/deleteByXX

ORM

Object Relational Mapping，对象关系映射。Relational指relational DB。包括Object->Relation和Relation->Object两方面。H是完整的ORM。

Hiberate VS Mybatis

* Hiberate：实现O/R映射，OO操作DB。优势：Dao层维护简单。OO和面向SQL操作配合使用，看具体场景。
* Mybatis：优势：简单易上手，从SQL入手好优化和解决问题。劣势：操作要写SQL，不管annotation还是XM，在追求零配置的今天都显简陋。

补充：H的面向SQL较M麻烦，还是要做O/R映射，根据case选择性用。

JPA

[Reference](https://blog.csdn.net/oChangWen/article/details/52788274)

Java持久化API，一种规范，充分吸收现有ORM框架的基础发展而来。本质上是一种ORM，由厂商实现（比如Hibernate）。

* ORM映射元数据映射：@Entity,@Table等注解。
* JPA API: 实体的操作。面向对象非面向DB，操作实体而不是db的table。
* 查询语言：

### 2.3.3 Spring Security

<input type=”password”> 密码虽然隐藏，可通过js获取，当然客户端只是防止偷窥，安全的重点是传输过程。

流程

form submit username/password (明文)🡪后台db存储加密的pass🡪check

实现原理

web.xml

1. **<filter>**
2. **<filter-name>**springSecurityFilterChain**</filter-name>**
3. **<filter-class>**org.springframework.web.filter.DelegatingFilterProxy**</filter-class>**
4. **</filter>**
6. **<filter-mapping>**
7. **<filter-name>**springSecurityFilterChain**</filter-name>**
8. **<url-pattern>**/\***</url-pattern>**
9. **</filter-mapping**

登录成功后user账号和sessionId关联，每次在filter中校验。

### 2.3.3 Bean生命周期

[Reference](https://www.zhihu.com/question/38597960)



* Bean instance
* BeanPostProcessor—postProcessBeforeInitialization
* InitializingBean-afterPropertiesSet
* Init-method：相当于@PostConstruct。
* BeanPostProcessor—postProcessAfterInitialization

为什么需要Init-method?

1. **public** **class** Car {
2. @Inject
3. **private** Engine engine;
4. **public** Car() {engine.initialize();  }
5. }

construct早于inject,这里调用egnine会导致nullPointException

作用域

singleton：生命周期同Spring Container，在容器中只有一个实例。

prototype：每次调用都会得到一个新的实例。

request, session, global session：以上三类是只有引入了Spring ApplicationContext才存在，生命周期大致对应HTTP中的Request, Session和Application。

TODO1:Spring管理的对象，通常是单例的吗？不然？为什么不都用单例的？

### 2.3.5 Spring Boot

简化应用配置，更容易使用spring。方便搭建项目或构建一个微服务。

* 简化配置：封装常用套件，比如mybatis、hibernate、redis、mongodb等
* 自动管理依赖。
* 部署简单：内嵌Web容器，如 Tomcat

缺点是集成度较高，使用过程中不太容易了解底层。

注解说明

[Reference](https://blog.csdn.net/javaloveiphone/article/details/52182899)

@Configuration 相当于xml中的<beans>，用于配置spring容器。

spring-boot 2.0

#### 微服务架构

随着微服务的概念也慢慢兴起，快速开发微小独立的应用变得更为急迫，Spring 刚好处在这么一个交叉点上，于 2014年发布Spring Boot 1.0.0。

#### Starter

Starter集成框架并提供缺省参数，是实现自动化配置的关键

spring-boot-starter-web

集成Spring MVC,Tomcat，Jackson等库

spring-boot-starter-data-jpa

基于hibernate的JPA框架。包含datasource，driver/事务管理器等。

Spring boot Actualtor

Actualtor比较全面的监控了Spring Boot应用的整个生命周期。

监控内容：应用状态、内存、线程、堆栈等等

查看方式：接口返回json

Spring Boot Admin

封装actuator接口，用UI页面显示，方便查看。可直接修改logger的level

package结构

com.example.myproject

+- Application.java

+- domain

+- service

+- controller

Spring 响应式编程

Spring WebFlux是Spring5的一个新模块，包含响应式HTTP和WebSocket 的支持，在上层服务端支持两种不同的编程模型：

* 基于 Spring MVC 注解 @Controller 等
* 基于 Functional 函数式路由

## 2.4 框架

### 2.4.1 Hibernate

ORM框架，实现POJO与db表的映射，SQL自动生成执行

对象三种状态

Transisent Objects：db不存在，session里也没有（比如new的对象）。

Persist Objects：save进入持久化状态🡪set属性,此时tx.commit直接保存到db。

Detached Objects：session.close()后，object有id，db里有对应数据。

使用方式

jdbcTemplate：使用简单，复杂功能需要自己封装。

缺点：

* 实现不了复杂查询/特定语法
* 写原生SQL，不能用缓存
* 注意的点太多，调优困难。

#### H Session

指会话，一个session对应一个connect，是种缓存机制 (结合Http的session理解，在server端保存client会话信息)

事务和session关系

事务在db层面实现，session关闭并不引起事务关闭

缓存机制

* Session级缓存: 应用场景太单一。通过ThreadLocal在线程中重用Session，单个线程可能处理不用业务，缓存命中率很低。大量的列表式select，缓存起不到作用。
* SessionFactory应用级缓存。EHCache是Hibernate中的二级缓存插件，使用Hibernate的系统可以直接使用EHCache缓存。

[Reference](http://coolvinson.iteye.com/blog/520610)

1. //1.获取sessionFactory
2. SessionFactory sessionFactory = HibernateUtil.getSessionFactory();
3. Session session = sessionFactory.openSession();//打开一个新的sesion
4. Transaction tx = session1.beginTransaction();
5. CaUser user = (CaUser) session1.get(CaUser.**class**, 1000L);
6. tx1.commit();
7. session.close();

* get/load 返回代理对象，延迟加载, getCurrentSession获取当前线程里的session，如果没有才打开新的

#### Hiberate事务锁

乐观锁

[Reference](http://www.voidcn.com/article/p-sdnzagtc-oy.html)

1. @Version
2. **private** **int** version;

H最终生成SQL

1. **update** CA\_USER **set** USER\_NAME=?,version=? **where** id=? and version=?

对于version已经改变的update,rowCount是0，如下图将throw StaleStateException。



系统报错，系统捕捉异常，转换友好的提示到UI层。

悲观锁

依靠数据库层面的锁机制，查询的时候对表数据进行锁定（实际应用中悲观锁是很少被使用的，因为它大大限制了并发性）。



Thread2执行时将阻塞到报错，因为t1加了排他锁，t2不能加锁，select..for update

Caused by: java.sql.SQLException: Lock wait timeout exceeded; try restarting transaction

Hibernate LockMode

LockMode.NONE ：有缓存用缓存，没缓存则从数据库读   
LockMode.READ ：直接从数据库读，不使用缓存数据   
LockMode.WRITE ：在insert update数据的时候，H内部使用的。   
以上3种均为Hibernate级别的锁，也就是缓存级别的锁。   
下面2种为数据库级别的锁：   
LockMode.UPGRADE：相当于SQL语句select for update，被select的数据都被数据库锁住了，不能被其他事务修改。   
LockMode. UPGRADE\_NOWAIT ：是ORACLE数据库特有的select for update nowait

加锁策略

如果每次访问冲突概率小于20%，推荐使用乐观锁，否则使用悲观锁。乐观锁的重试次数不得小于3次。

### 2.4.2 Mybatis

本质上同JDBC，都是面向SQL操作DB 。但Mybatis完成SQL🡪Object(比如框架设置参数和封装返回结果)的Sql Maping，从而简化JDBC操作

历史：由Apache在2001年发起，2010年由Google托管改名MyBatis。

像阿里等很多互联网公司用的多也是因为

* 方便排查问题，直接从sql入手解快，缩短定位问题要经历的栈（H需要层层找问题)，
* SQL好优化，加上团队H使用水平参差不齐

#### 映射

xml or annotations

#### Annotations

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNDMwMTMwMw==&mid=2247490260&idx=1&sn=bf9bda99c51d18c96a0171480d564ba2&chksm=9b943bccace3b2daed00fd61ff362b19735fcaa85e16655c4069f5dd1fe06eb65159ad7fd94c&mpshare=1&scene=1&srcid=0111qEOitcRE9X6xqIeygwov#rd])

基本注解，**CRUD：**@Select@Insert@Update@Delete

1. @Select("select name, user\_id from t\_user where user\_name = #{userName}"
2. User getUserByName(@Param("userName") String userName);

驼峰映射完成sql和entity的映射。若属性和字段驼峰不一致@Results,@ @ResultMap

高级注解，主要用于动态SQL。@SelectProvider @InsertProvider @UpdateProvider @DeleteProvider

全局配置文件mybatis-config

[Reference](http://www.mybatis.org/mybatis-3/zh/configuration.html)

settings

1. **<settings>**
2. **<setting** name="mapUnderscoreToCamelCase" value="true"**/>**
3. **</settings>**

DB连接池

建立连接是一个费时的操作，频繁的进行数据库连接操作势必占用很多的系统资源，影响网站的响应速度。

如果查询完数据库后，不关闭连接，而是暂时存放起来，当别人使用时，把这个连接给他们使用。就避免了一次建立数据库连接和断开的操作时间消耗。

基本思想

为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去

TODO3:C3P0源码

### 2.4.3 Mina

Netty&Mina：对JDK网络编程的封装，屏蔽了繁杂的编程细节，提供便于用户开发网络应用程序的api,让开发者可以更加专注于业务逻辑的实现。

NoSql/redies：如何使用NIO的

Tomcat NIO VS Netty

最大的区别就在于通信协议，Tomcat是基于Http协议的web容器，Netty可自定义协议，因为netty能够通过codec自己来编码/解码字节流，完成类似redis访问的功能，这就是netty和tomcat最大的不同。

netty的性能不一定就一定比tomcat性能高，tomcat从6.x开始就支持了nio模式，并且后续还有arp模式，相比于旧的bio模式，并发性能得到了很大提高，特别是arp模式。

#### 连接管理

session相对http较长,手动管理。

1. **public** **class** SessionManager{
2. Map sessionMap= **new** ConcurrentHashMap<Long,IoSession>();
3. addSession (Long id,IoSession session);
4. removeSession (Long id,IoSession session)
5. }

#### 心跳机制

监控无效的连接并断开

* TCP协议自带的keep-alive功能,打开即可
* 应用层面实现心跳机制，自定义心跳消息头. 一般客户端主动发送。服务器接收后进行回应(也可以不回应)。
* 框架自带，MINA本身提供了一个过滤器类[KeepAliveFilter](http://mina.apache.org/mina-project/apidocs/org/apache/mina/filter/keepalive/KeepAliveFilter.html)

长连接&心跳机制

1）TCP协议自带的keep-alive功能,打开即可

1. Socket socket=serverSocket.accept();
2. socket.setKeepAlive(**true**)

当keepAlive为true，server定时发送数据包检查客户端的状态，没response将socket关闭。

3）框架自带

MINA本身提供了一个过滤器类[KeepAliveFilter](http://mina.apache.org/mina-project/apidocs/org/apache/mina/filter/keepalive/KeepAliveFilter.html)  ,在IO空闲的时候发送并且反馈心跳包

1. KeepAliveFilter heartBeat = **new** KeepAliveFilter(heartBeatFactory);
2. heartBeat.setForwardEvent(**true**);//是否回发
3. heartBeat.setRequestInterval(HEARTBEATRATE);//发送频率
4. //connector.getSessionConfig().setKeepAlive(true);
5. conn.getFilterChain().addLast("heartbeat", heartBeat);

#### codec

自己来编码/解码字节流

#### 工作原理

架构多工作线程Reactor模式



Acceptor：处理连接请求

**NioProcessor：**循环读



### 2.4.4 Netty

对JDK网络编程的封装，屏蔽了繁杂的编程细节，提供便于用户开发网络应用程序的api,让开发者可以更加专注于业务逻辑的实现。

基础通信组件,RPC工具

很多中间件把netty作为基础通信组件。阿里分布式服务框架 Dubbo ,淘宝的消息中间件 RocketMQ

**1.异步**

非阻塞，事件驱动机制

**2.高性能的通信框架**

不用依赖于容器去进行部署,易扩展

## 2.5 中间件

中间件提供OS,DB之外的服务，不是软件应用的一部分，让开发者专注于业务开发。

### 2.4.1 MongoDB

特性

* 数据模型：no schema,其结构是面向对象的而不是二维表。使用灵活
* 易伸缩：指的是分片能力，
* 故障转移：自动检测主节点是否存活，失活时自动提升从节点为主节点，达到故障转移。

使用场景

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247485536&idx=1&sn=dd24c29beb691b1efe4544a9a7c43a4a&chksm=96cd482ca1bac13a33031815b43b8d0c814f8d2998a43b4218bbe0d18ebc352b6d09af805b6f&mpshare=1&scene=1&srcid=&pass_ticket=cW2JCTQ4rkqDxvQI35OMgWpyjAos1512GIFXJP3ta6rRXyKD1LcWRqRcSA3RsKxL#rd)

大部分mongodb使用场景，mysql也能解决，但通常能以更低的成本解决问题（包括学习、开发、运维等成本)

案例1 服务器的日志记录，查找起来比文本灵活，导出也很方便。

案例2 存储监控数据，No schema ，增加字段不用改表结构，很方便。

Nosql

现代互联网应用，关系数据库的很多主要特性却往往无用武之地

1.数据库事务一致性需求

2.数据库的写实时性和读实时性需求

3.对复杂的SQL查询，特别是多表关联查询的需求

去掉关系型数据库的两大重要基础：以关系代数为基础的结构化查询语句（SQL）和事务一致性保证（ACID）

1. 高并发读写2)海量数据的高效率存储和访问3)高可用性和可伸缩性

### 2.4.1 Redis

为什么使用？

主要考虑两点：性能和并发

* 高性能：缓存数据，请求直接访问缓存，减轻db压力，加快响应速度。
* 并发：采用非阻塞I/O多路复用机制,支持更高并发
* 实现分布式锁（可用Zookepeer等其他中间件代替）。

分配KEY：分配key的时候，它会使用CRC16算法属于哪个槽(集群裂变16384个hash slot)

memcache VS redis

* redis单线程支持更高的并发
* redis集群功能强大，分片应当海量数据

数据一致性：Memcached提供了cas命令来保证.而Redis提供了事务的功能。

* redis数据可以持久化

单线程为什么这么快？

[Reference](https://mp.weixin.qq.com/s?__biz=MzI4NDY5Mjc1Mg==&mid=2247485753&idx=1&sn=80faf2588a83db267cf72fb2d29372ce&chksm=ebf6d146dc815850073311655374dfe71c3efb03631e799938e876ef3c2ac3587c3801f44fcc&mpshare=1&scene=1&srcid=0918VwJdE9EfWL3CvPHMJPLi#rd])

* 纯内存操作
* 采用非阻塞I/O多路复用机制，支持更高的并发。
* 单线程，避免频繁上下文切换。

数据类型

问题

缓存雪崩：缓存同一时间大面积失效，这时又来一波请求。（缓存时间用随机值）

缓存击穿：故意请求缓存中没有的数据。导致请求都落得DB，导致db连接异常。（缓存失效访问db采用互斥锁，控制db的访问）

Redis 和数据库双写一致性问题

数据库和缓存双写，如果对数据有强一致性要求，不能放缓存。所做方案，只能降低不一致发生的概率。

首先，采取正确更新策略，先更新数据库，再删缓存。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

#### 架构分析

扩展性：无中心节点，方便线性扩展性。

集群

Redis集群是构建高性能网站架构的重要手段。

[Reference](https://juejin.im/entry/596343056fb9a06bc340ac15)

master-slave: master宕机，slave只能读，不可写，不能保证高可用

主从复制

redis支持master-slave模式，一主多从，减轻主机的压力。

防止单点故障。cluster每个节点设计为主从复制模式。

一致性：保证最终一致性。主从复制过程，可能会出现数据延迟和丢失。注意业务场景。

sharding

方便从海量数据根据条件查询。Redis集群使用数据分片而非一致性哈希实现。

### 2.4.2 Memcached

并发的处理

1.单个命令是原子操作。命令将被串行化、先后执行。

2.命令序列不是原子的。在并发的情况下，get and set可能覆写其他进程set的item。Memcached 1.2.5以及更高版本，提供了gets和cas命令，可解决。

JAVA客户端

1. Memcached client for java (code: MemCache120)

官方提供, 运行比较稳定，使用阻塞IO，不支持CAS操作。

1. XMemcached

Web service

WebService，顾名思义就是基于Web的服务。基于HTTP传输协议的程序。接收和响应外部系统的某种请求。从而实现远程调用。

Solr

页面搜索严禁左模糊或者全模糊，如果需要请走搜索引擎来解决

## 2.6 应用服务器

TODO1:为什么大多数的服务都运行在Linux

TODO2如何做压力测试出负荷值

### 2.6.1 Tomcat

遵循Servlet规范，又叫Servlet容器。在中小型系统和并发访问用户不是很多的场合下被普遍使用。

版本历史

* 6.0.x (2007) Jdk5 Servlet2.5
* 7.x (2011) Jdk6 Servlet 3.0，部分支持WebSocket(标准未定).
* 8.x (2013) Jdk7 Servlet 3.1；支持NIO2，Java WebSocket 1.0
* 8.5 (2016) 支持HTTP/2
* 9.x (beta) Jdk8 Servlet4.0; JSP2.4 ;EL3.1

#### 支持协议

BIO，NIO和APR。到Tomcat8.5，BIO模式因为效率太低已被淘汰。

APR：是Apache HTTP服务器的支持库。可以简单地理解为Tomcat将以JNI的形式调用Apache HTTP服务器的核心动态链接库来处理文件读取或网络传输操作，从而大大地提高Tomcat对静态文件的处理性能

#### 性能优化

Tomcat的默认配置作为生产环境，尤其是内存和线程的配置，默认都很低，容易成为性能瓶颈。 可以从三方面入手，内存/线程/IO

线程

默认有12个系统线程负责。。。

Thread busy: jstack pid | grep " RUNNABLE" | wc –l

waitting: jstack pid | grep "WAITING" | wc –l

如果tomcat线程池minSpareThreads设置较大，大量线程将parking

* maxThreads：Tomcat7默认值200。线程用于计算的时间可能很少，大多数线程都在等待或阻塞(如等待数据库返回数据、等待硬盘读写数据等)。maxThreads应设大点，能支持更高的并发，最大化利用cpu闲置资源。但并不是越大越好，过大CPU会花费大量的时间用于线程的切换，整体效率会降低。
* minSpareThreads：初始创建数。默认25。
* maxSpareThreads： max，超过将关闭socket。默认值75。
* maxConnections：可以处理的连接数目。BIO：maxConnections=maxThreads；NIO：maxConnec应该远大于maxThreads
* acceptCount：OS级别的处理请求队列值。默认值10。Server可同时接收的连接数为maxConnections+acceptCount 。acceptCount的设置，与应用在连接过高情况下希望做出什么反应有关系。如果设置过大，后面进入的请求等待时间会很长；如果设置过小，后面进入的请求立马返回connection refused
* acceptorThreadCount： acceptor数量

KeepAlive

maxKeepAliveRequests： 保持请求数量，默认值100。

keepAliveTimeout 默认值60000

keepAliveTimeout VS connnectionTimeout

connnectionTimeout： 网络连接超时，默认值60000，单位：毫秒。通常可设置为30000毫秒。设置为0表示永不超时，这样设置有隐患的

bufferSize： 输入流缓冲大小，默认值2048 bytes。

compression： 压缩传输，取值on/off/force，默认值off。 其中和最大连接数相关的参数为

enableLookups： 是否反查域名，默认值为true。为了提高处理能力，应设置为false

server.xml

1. **<Connector** port="8080" protocol="HTTP/1.1"
2. connectionTimeout="20000" maxThreads="1000" minSpareThreads="60"
3. maxSpareThreads="600" acceptCount="120"  redirectPort="8443" URIEncoding="utf-8"**/>**

内存

[Reference](https://mp.weixin.qq.com/s?__biz=MzI4NDY5Mjc1Mg==&mid=2247486845&idx=1&sn=618c05eedd0a0367870055aff8195912&chksm=ebf6d502dc815c14dd881e362e2f04e59a8bce965324be584abbee55f50712984bfc961575f1&mpshare=1&scene=1&srcid=#rd])

#### 实现原理

[Ref Demo](https://www.jianshu.com/p/dce1ee01fb90)

Connector

1)提供Socket服务。

2)进行请求的分发

3)把请求和响应封装成request/response

HttpRequest🡪Acceptor🡪Poller🡪Worker🡪NioProcessor

Acceptor

1. SocketChannel sc= serverSocket.accept();
2. PollerEvent event= **new** PollerEvent(sc);
3. queue.offer(event)

Poller

1. PollerEvent event= queue.poll();
2. threadPool.execute(**new** Worker(event));

Worker

1. **class** Worker **implements** Runnable{
2. HttpServletRequest request = read(socket);
3. Servlet servlet= ServletContainer.get(request);
4. servlet.doGet(request);
5. }

### 5.3 Nginx

优点

* 异步非阻塞，高并发环境消耗更少的资源。高并发高吞吐量的Web服务。
* 使用方便，配置简洁。

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247484647&idx=1&sn=cecb4382b57c2fc7db868e71a2a277ad&chksm=96cd44aba1bacdbdbe3e8aa9b070a5ddd83d587847075f6edc97b58602d7491a6ce17244c6f6&mpshare=1&scene=1&srcid=1010IsfkdqydT8XpHNZQpuU6#rd])

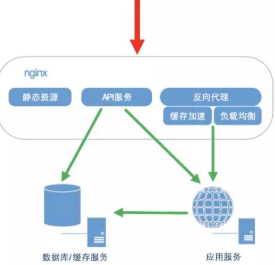
配置

1. upstream cluster1 {
2. server localhost:8080;
3. server localhost:8081;
4. }
6. server {
7. listen       81;
8. server\_name  localhost;
9. client\_max\_body\_size 1024M;
11. location /home {
12. proxy\_pass http://cluster1;
13. proxy\_set\_header Host $host:$server\_port;
14. }
15. }

上面upstream默认负载均衡测率为RR

#### 使用场景

负载均衡+反向代理



1) Htttp服务器

前后端分离，动静资源拆分后，将静态资源做缓存操作。

1. # 所有静态请求都由nginx处理，存放目录为html
2. location ~ \.(gif|jpg|jpeg|png|bmp|swf|css|js)${
3. root    e:\wwwroot;
4. }
6. # 所有动态请求都转发给tomcat处理
7. location ~\.(jsp|do)$ {
8. proxy\_pass  http://test;
9. }
10. error\_page  500 502 503 504 /50x.html;
11. location =/50x.html {
12. root   e:\wwwroot;
13. }
14. 代理

正向代理：代理客户端，代理客户端发送请求，如VPN。反向代理：代理服务器，代理服务端处理请求，如nginx

反向代理

代理的基础上可以实现负载均衡或缓存加速

缓存加速

Nginx的缓存加速功能是由proxy\_cache（用于反向代理和静态缓存）和fastcgi\_cache（PHP动态缓存）两个功能模块完成

正向代理

3）API服务

OpenRestry

#### 负载均衡

Nginx负载均衡策略大致分为内置策略和扩展策略

内置策略：轮询、ip\_hash、最少连接

扩展策略：fair、url hash等

1.轮询(RR)

Round-robin。默认方式。按请求顺序，逐一分配。自动判断server状态，忽略down掉的server。

Session要做共享。

2.ip hash

原理： hash(ip)%server num，余数即server号。保证请求转发到特定的server，session保存在一台服务器，不做session共享。

相同ip的请求将永远落在同一台服务器上，这为均衡性埋下了较深隐患。

1. upstream cluster1 {
2. ip\_hash;
3. server localhost:8080;
4. server localhost:8081;
5. }

3.加权轮询

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。

1. upstream test {
2. server localhost:8080 weight=9;
3. server localhost:8081 weight=1;
4. }

4.fair

fair策略是扩展策略，它根据后端服务器的响应时间判断负载情况，从中选出负载最轻的机器进行分流。这种策略具有很强的自适应性，但是实际的网络环境往往不是那么简单，因此须慎用。

5.url hash

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。

Apache 做负载均衡，如何实现session共享?

* session复制
* session sticky 用户的请求转发到特定的Tomcat服务器上

Nginx与apache

都是Http服务器。Nginx基于NIO，支持高并发，性能高。而apache是阻塞的。但比较稳定。

* 动静态资源分离：反向代理分发请求：动态资源的请求交给Tomcat，而静态资源的请求直接由代理返回到浏览器。
* 负载均衡：降低单点Tomcat压力,提升响应速度。实现水平扩展。

### 5.4 Jetty

## Web设计实践

或者叫常见的解决方案

### API设计

需求：Desktop时代前端后端融合在一起（JSP），随着互联网发展，各种类型client层出不穷。因此需要一种统一的机制，方便不同的前端设备与后端进行通信。这导致API构架的流行，RESTful API是目前比较成熟的一套互联网应用程序的API设计理论。

#### 接口设计

1）设计原则

开放API接口设计没有什么特别，原则同上篇OO设计准则

* 单一职责：粒度适中，也不要做多职能接口。
* 开闭原则：接口要做到开闭，避免大批量下游不可用。同时也要避免过度设计，当抽象功能只有一处使用时，尽量不要过早抽象。
* 命名规范，文档清晰详细。返回值要友好

2）同步 VS 异步

3）RPC VS NoRPC

REST: 跨语言，只需根据url传参调用。服务独立发布部署。耦合度低

第三方jar： 直接调用性能佳，但升级麻烦，

4）RPC VS REST

RPC: REST API 和 RPC 都是函数封装成接口暴露出去，以供调用。不过 REST API 一般都是基于 HTTP 协议，而 RPC 则可以是socket,http,SOAP等.Rest是web service的一种方式,可以归到Http协议的RPC。

5）高并发

#### RESTful 风格

[Reference](http://www.ruanyifeng.com/blog/2011/09/restful.html)

REST特点：结构清晰、符合标准、易于理解、扩展方便

设计细则

[Reference](http://www.ruanyifeng.com/blog/2014/05/restful_api.html)

URL：<https://api.example.com/v1/zoos>

* 协议：API与用户的通信协议，总是使用HTTPs协议。
* 域名：尽量将API部署在专用域名之下。
* 版本：API的版本号放入URL。
* 路径：路径又称"终点"（endpoint），表示具体资源。所以只能是名词，且为复数。
* Http动词：对于资源的具体操作类型，由HTTP动词表示。GET/POST/PUT/PATCH/DELETE
* Filtering：允许路径和URL参数偶尔有重复。比如,/z/1/a 与/a?z\_id=1。
* Status Codes：返回的状态码和提示信息，200 OK，201 CREATED
* Error handling：如果状态码4xx，应返回出错信息{error: "Invalid API key"}
* Hypermedia API：hypermedia-driven，不用看文档，使用方便。

3.3.4.1 RESTful架构

Representational State Transfer，表现层状态转化，是一种互联网软件架构，Client和Server的交互性形式。表现层指的是"资源"（Resources）的"表现层"。

Resources

"资源"，是一个信息实体。可以是一段文本/一张图片/一种服务。每种资源对应一个特定的URI。要获取这个资源，访问它的URI即可。

表现层（Representation）

资源有多种外在表现形式。具体呈现出来的形式叫做它的表现层。比如，文本可以用txt/XML/HTML/JSON表现。

State Transefer

通过Http动词实现。

REST 成熟度模型

**Level 0：** 使用 http作为传输方式。

**Level 1：** 引入了资源的概念。

**Level 2：** 使用 http 方法进行操作，并使用 http状态码来表示结果。

Level 3：使用HATEOAS。在资源的表达中包含了链接信息。client根据链接来发现可执行的动作。

Hypermedia API

[Reference](https://spring.io/understanding/HATEOAS)

hypermedia-driven,make it easy to glean how to interact with the server without looking up a specification or external document。

比如集合接口的每个item都有href连接，图片接口返回不同大小图片的链接

#### 前后端分离

1.随着移动应用的快速发展，此时出现前后端分离。

当前趋势网络应用程序应分为前端和后端两个部分。

### 异常设计

[Reference](https://www.jianshu.com/p/503901e5be2b)

减少程序错误，提高健壮性(Exception可处理)。是重要的调试手段，根据抛出的异常能判断发生的位置和出了什么出错，为什么出错。

什么时候抛异常？

可能会产生问题就抛出异常。

#### Java异常



Error VS Exception

* Error：不可处理，编译和系统的错误，应中断。OutOfMemoryError
* Exception： 可处理。

Checked Exception

强制catch,否则编译不通过。比如IOEx/ClassNotFoundEx。

Problems

处理麻烦，可读性差，大部分情况coder不知道如何处理，强制处理不现实（保证兼容性不可去掉），Think in Java建议:

1.print to console

2.转换成unchecked抛出保证信息不丢失。

#### 什么时候抛异常

可能出现问题的地方

1. Address createAddress(Integer uid, Address address) {
2. Preconditions.checkNotNull(uid);
3. User user = userDao.findOne(uid);
4. **if**(**null** == user)    **throw** **new** RuntimeException("找不到当前用户!");
5. }

业务校验时可以抛异常。

#### 系统异常处理

一般出现异常记录日志后逐层向上抛。在web层设置统一异常处理机制。

api：应用内部推荐异常抛出，跨应用/开放API接口要返回错误码和提示信息。

页面：要跳转到友好错误页面。

1. **public** **class** CustomExceptionResolver **implements** HandlerExceptionResolver {
2. MV resolveException(request,response, Object handler, Exception e) {
3. LOG.error(e.getMessage(),e);
4. ModelAndView mv = **new** ModelAndView();
5. **if**(e **instanceof** AppException) {
6. String msg={"success":false,"msg":+e.getMsg();
7. response.getWriter().write(msg);
8. }**else** **if**(e **instanceof** CustomException) {
9. mv.addObject("message", e.getMessage());
10. mv.setViewName("/WEB-INF/jsp/error.jsp");
11. }
12. **return** mv;
13. }
14. }

将异常分类

一般异常框架有如下几种分类

1）AppException

只用一个AppException，在这个异常类中添加一个type属性，通过type来区分不同的异常。type可以用一个枚举来维护，type值可以是参数格式、业务校验、服务。也可以只用一个AppException，所有的异常都封装成这个，不再按type细分。

2）按层定义

DaoException/ServiceException/AppException。使用时service层把DaoException转为ServiceException，再在应用层把ServiceException封装成AppException。最外层有一个拦截全局异常，拦截AppException就可以了。

4）分类

ArgumentException: 参数格式异常，在所有的参数格式校验出错的时候使用。

BusinessException: 业务校验异常

ServiceException: 服务异常

1）设计的很细

异常太多，太麻烦。比如User表格字段校验异常。

自定义异常

DAO层和Service层，产生的异常类型有很多，无法用细粒度的异常进行 catch，避免直接throw new Exception,应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException等。

使用技巧

3.代码可读性，干掉if else，减少嵌套?

1) 仅异常情况下使用异常

异常不要用来做流程控制，条件控制。异常设计的**初衷**是解决程序运行中的各种意外情况，且异常的处理**效率**比条件判断方式要低很多。



（虽然这里的异常可以用return Result，来减少嵌套并携带错误信息）

2) 异常带上参数信息

记录日志时，尽可能带上参数信息，便于调试

3) 不要捕捉所有异常

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (Exception e) {  }

不推荐的写法，应尽量避免,这样些会可能带来一些问题：

* 调用的接口的潜在Bug被忽略
* 不同类型的异常可能需要不同的处理，写在一起内聚性比较差

上面的代码可以更改为：

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (IOException e) {
5. // Do something handle IOException
6. } **catch** (CertificateException e) {
7. // Do something handle CertificateException
8. }

如果所有异常处理方式一致，且JDK版本高于1.7，可以简化如下：

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (IOException | CertificateException e) {
5. // Do something handle these exception
6. }

### 菜单递归查找

Find all root menu

1. **public** **void** buildMenu() {
2. List<Menu> allMenus = **new** ArrayList<Menu>();//dao.getAllMenus();
3. List<Menu> parents = **new** ArrayList<Menu>();
4. //find all 1 level menu
5. **for** (Menu menu : allMenus) {
6. **if** (menu.getParentId() == **null**) {
7. parents.add(menu);
8. }
9. }
10. **for** (Menu menu : parents) {
11. menu.setChildren(getChild(menu.getMenuId(),allMenus));
12. }
13. }

getChild

1. **public** List<Menu> getChild(**long** menuId,List<Menu> menuList) {
2. List<Menu> childList = **new** ArrayList<Menu>();
3. **for** (Menu menu : menuList) {
4. **if** (menu.getParentId().equals(menuId)) {
5. childList.add(menu);
6. }
7. }
8. //递归终止条件
9. **if** (childList.size()==0) {
10. **return** **null**;
11. }
12. //开始递归
13. **for** (Menu menu : childList) {
14. menu.setChildren(menu.getChild(menu.getMenuId(),menuList));
15. }
16. **return** childList;
17. }

### 应用分层

1.结构清楚，2.分工明确，提升可读性，好维护和升级。3.降低模块间的耦合度。

SpringMVC

Spring 框架已经成为构建企业级 Java 应用事实上的标准了，很多遵循着如下分层：Controller🡪Service（业务逻辑）🡪Dao（Data Access Object）

MVC的M = Service+Dao，将业务逻辑和db访问分开。

2.1.3 MVC

1978年提出，MVC不是一种技术，而是一种理念，强调职责分离。

背景：早期JSP网页，SQL query和HTML代码混在一起。MVC从根本上强制分层，尽管需要一些额外的工作，但好处是毋庸置疑。

MVC是一种软件设计模式，MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

**View**: UI操作界面，Model包含数据的显示。

**Model**：封装数据源和对这些数据的操作。

**Control**：介入view和model，处理user在view上的输入，并转发给model，这样Model和View两者之间可以做到松散耦合，甚至可以彼此不知道对方，而由Controller连接起这两个部分。

好处

便于分工合作。

降低系统复杂度。

todo SPRING ModelAndView

更细致的分层

在阿里的编码规范中约束的分层如下:



* **终端显示层**：PC端，移动端展示等。
* **开放接口层**：封装 Service 方法暴露成 RPC 接口/通过 Web 封装成 http 接口（需要进行网关安全控制、流量控制等）。
* **Web 层**：即Controller层，主要是各类基本参数校验，或者不复用的业务简单处理等。
* **Service层**：相对具体的业务逻辑
* **Manager 层**：通用业务处理层，它有如下特征：

1）对第三方平台封装的层，预处理返回结果及转化异常信息；

2）对 Service层通用能力的下沉，如缓存方案、中间件通用处理；

3）与 Dao 层交互，对多个 Dao的组合复用。

常规校验

在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放限制，

如数量限制、疲劳度控制、验证码校验，避免被滥刷、资损。

说明： 如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其它用户，并造成短信平台资源浪费。

参数校验

用户请求传入的任何参数必须做有效性验证。

说明：忽略参数校验可能导致：

page size 过大导致内存溢出

恶意 order by 导致数据库慢查询

任意重定向

SQL 注入

反序列化注入

正则输入源串拒绝服务 ReDoS

说明：Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，

但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

**参数的安全性判断应该写在controller还是service？异常处理尼？**

哪一层都是很有必要的，一般在controller进行，在service层进行校验更好的复用。

### Session

Session用来在Server端保存临时全局保量，Session+Cookie：弥补Http无状态的不足。

Session VS Token

Token：Server端Session的存储和查询都要占用一定的系统资源。同Session，client的每个请求都要携带。[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNDMwMTMwMw==&mid=2247490367&idx=1&sn=3a3f62570435ef1808369676c707a00c&chksm=9b943a27ace3b3316108b3833301fc10e9bca0e4bf83c928fae024ad52384c44d38791ded619&mpshare=1&scene=1&srcid=#rd])

原理：以User登录为例，Session在服务端需要记录用户的状态，Client在Cookie且在每次请求带上cookie信息，server根据请求的cookie验证登录状态。

TODO:Session劫持

#### 实现原理

在服务端保存session的方法很多(内存、数据库、文件等)。Servlet容器的Session数据结构为Servlet规范中的HttpSession实现类，由厂商实现。**以Tomcat为例**。

1.服务端在Client第一次request生成的JSESSIONID放内存，并写到客户端cookie，

2.Client之后的request的head都带着sessionId。

TODO1 session复制如何配置？

（如果不设置cookie过期时间，cookie会在会话结束后销毁，称为会话cookie）

问题隐患：

sesssion存在server内存，增加服务端压力。

#### 集群session

集群的时候要考虑Session的共享。

1）存Cookie

原理：client每次请求带着完整cookie信息。

缺点：增加网络负担。Cookie大小限制。

2）Tomcat Session复制

原理：每个节点存在相同的Session，新加入的复制一份Session信息。适用小规模的系统。

缺点：1.一致性问题。节点越多，损耗越大（延迟，宽带）。2.单机内存有上线。

1. Session sticky

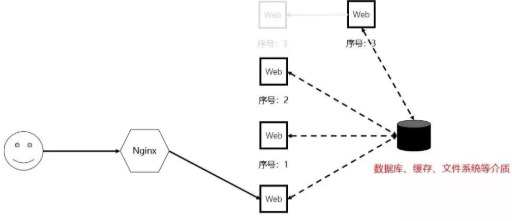
Apache做负载均衡器，用户的请求转发到特定的Tomcat服务器上。

缺点：服务器重启，丢失会话，可用性不高。

1. 集中管理

Session集中管理。

缺点：增加网络开销。



Session服务器方便横向扩展。可无限扩张，适用于分布式系统。

集中管理的具体实现：

1.spring session

2.实现 HttpServletRequestWrapper

#### Spring Session

[Reference](http://blog.csdn.net/xiao__gui/article/details/52706243)

设计一个Filter，利用HttpServletRequestWrapper，实现自己的 getSession()方法，接管创建和管理Session数据的工作。

Nginx + Redis

自定义filter, Nginx + tomcat+ redis

Nginx配置为non-sticky运行模式，也即每一个请求都可以被分配到集群中的任何节点

WebSocket session

#### 单点登录(SSO)

session管理

[Reference](http://www.cnblogs.com/ywlaker/p/6113927.html)

Unicode编码

### Web容器

Netty版 [Reference](https://mp.weixin.qq.com/s?__biz=MzI3NzE0NjcwMg==&mid=2650123218&idx=2&sn=649438ed0aa4fa681ea077dc0b3a8395&chksm=f36bb6f3c41c3fe5de8c1bfd5fddbebe49a6b87f334b904fa6232fad09fed91fc61d5dfe12d7&mpshare=1&scene=1&srcid=&pass_ticket=rMfcWI31qoBc%2BJleH2gnwJiWltH0srLHo3%2BPGnyWlwrmp8k7vmb0%2FxHQPasaYnLj#rd)

### 网站安全

Web Security

* input黑点：password input黑点可用js获取到值，在客户端仅仅用于防偷窥。

传输方式

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNDMwMTMwMw==&mid=2247489948&idx=1&sn=b71b3e5070cbba089f594a0425255c23&chksm=9b943884ace3b1924a7b0fd3b5521e016e531a180a44fb1e40bd319aa3386c89f323eff60e39&mpshare=1&scene=1&srcid=0111WJPmhORp4e0NDeFpoB0u#rd])

* 明文传输：Spring Security本质就是明文，配合Https，对代码是透明的。
* 对称加密：前端js加密。缺点：逻辑容易被破解。比如HTTP Basic Authentication，即便不可逆MD5(pass)。
* HTTPS非对称加密：中间人截获密文，伪造请求，同样可以攻击（窃取数据或转走余额）。

#### Spring Security

实现原理：

#### RESTful API

1.客户端HTTP请求登录

2.server鉴权通过返回token，并将token和客户端关系维护在redis或DB

3.client将 token保存在本地，之后放请求的header，一旦token过期再次请求。

### 电商网站

#### 基本功能

User：注册/管理/维护

Product：列表，管理

Order:下单/管理订单

#### 提交订单

重复提交 [Reference](https://www.jianshu.com/p/e618cc818432)

#### 支付流程

## 附件1 效率工具

### guava-libraries

### commons



Workspace：工作区

Index / Stage：暂存区

Repository：仓库区（或本地仓库）

Remote：远程仓库

入门教程：[Reference](https://mp.weixin.qq.com/s?__biz=MzI2OTQ4OTQ1NQ==&mid=2247485703&idx=1&sn=69d22b19f48f6bb3cecec163f4e946eb&chksm=eadecd47dda944517f9060c59952759ad65f605adba53a2ea2b34bce1058252f799de59d9ce7&mpshare=1&scene=1&srcid=#rd])

[Reference](https://blog.csdn.net/mine_song/article/details/70770467)



全红：删除 全绿：增加

**-p 详细包括diff内容**

git log -p <filename>查看某个文件的修改历史

**-n**

显示最后的几次提交日志

git log -2

git log –p -2 查看最近2次更新内容

**show**

git show <commit-hash-id>查看某次commit的修改内容  
git show [--stat]

查看本地修改了还未提交

git diff filename

权限

查看key

cd ~/.ssh

Git VS Svn

svn更像一个文件服务器+版本控制，集中式管理文件。Git是一个分布式版本控制系统

1. 分支管理

Git擅长的是程序代码的版本化管理。

场景一:在分支开发新功能，需要新建分支修复紧急的bug

svn:创建branch--> 将代码下载到本地--> eclipse打开项目

git :创建branch-->同一工作目录下切换分支

svn下载的是完整代码库，速度慢，特别对于大型的项目，文件数达上万个以上，将非常的耗时，而git下载的是文件的镜像，大文件无需下载，而且当打开项目时需要编译，配置开发环境。像大型项目整个库的代码本地硬盘都放不下。git只下载镜像文件，需要时才下载。

2.离线提交

为了不让开发的过程代码影响其他人，svn一般task完成时一次性提交。而git在开发过程可以自由的提交，并且只有自己可见，实现本地的版本管理，非常方便。

3.分支合并

git rebase 和 merge 有什么区别？**M(Ali)**

### Swagger

构建强大的RESTful API文档

注解方式生成API文档。

* 书写方便（springboot整合）
* 快速构建，提升效率
* 方便接口测试

### Lombok

[Reference](https://juejin.im/post/5a6eceb8f265da3e467555fe)

@Builder 属性对外保持private setter，而对属性的赋值采用Builder的方式，这种方式最优雅，也更符合封装的原则，不对外公开属性的写操作！

Todo Java日志系统

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNDMwMTMwMw==&mid=2247490951&idx=1&sn=268d053572a526bcec3415e6ba5865c1&chksm=9b943c9face3b589af7eed94b61ca590fabfc6e4817382def2cf65196280d59d2aa47a934b46&mpshare=1&scene=1&srcid=#rd)

# 3.7.2 SpringCloud

（源于2014年的一篇文章“Microservices”，本章内容将基于SpringCloud框架体系讲述）

### 3.7.1 什么是微服务？

微服务架构是一种架构模式，强调彻底的服务化和组件化。一个组件就是一个产品，可独立对外提供服务,独立运行空间，独立的DB。微指组件拆分粒度更小。

有什么好处

* 拆分力度更小更彻底，单一职责，分工明确。
* 微服务架构带来更好的扩展性，可靠性等

微服务架构从互联网企业兴起的，互联网企业特点：

1)用户规模大（并发量高数据量大）。2)发布/迭代周期短（不同子系统发布周期不同）。

微服务 VS SOA

不同时代的产物，要解决的具体问题不同。但没有本质的区别，并且微服务是由SOA演化过来的。在SOA用于服务治理，提高集群资源效率。微服务架构从互联网企业兴起的。解决出现的新问题。

[微服务架构，多“微”才合适？](https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651961970&idx=1&sn=60631fc64b42bff46dcf2bc4eab60b13&chksm=bd2d0fae8a5a86b87861b5cf71abb334d42e8cff3315f13dc5aac560c2c350e425fbd8bdfd90&mpshare=1&scene=1&srcid=&pass_ticket=cW2JCTQ4rkqDxvQI35OMgWpyjAos1512GIFXJP3ta6rRXyKD1LcWRqRcSA3RsKxL#rd)

特点

从下面特点感受微服务是什么样子

* 通信轻量级：通常基于HTTP的RESTful API
* 技术异构：具体到一个服务，根据需要选择不同语言、工具。
* 复杂度：不同于接口直接调用，通过REST需考虑故障，过载，消息丢失等

微服务治理框架，基于Spring Boot开发。

核心组件

Eureka：用于服务的注册于发现。

Feign：服务的调用以及均衡负载。

Hystrix：处理服务的熔断防止故障扩散。

Spring Cloud Config：服务配置。

Gateway：服务网关

支撑服务

Dapper/zipkin+Sleuth：服务链路追踪

ELK：日志聚合。Elasticsearch+Logstash+Kibana

Spring Boot Admin：服务监控。

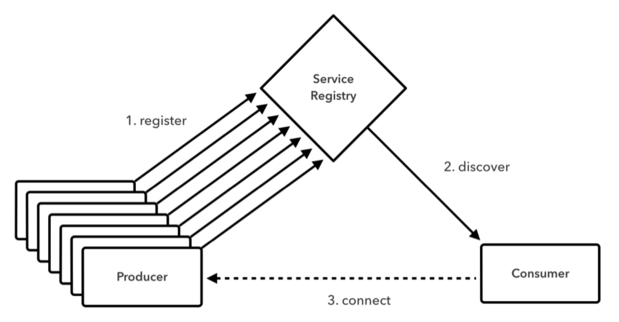
Docker：部署发布。

项目根据实际需求和业务场景，选择使用到的组件。

#### Eureka

分布式环境，实现服务的注册和发现，满足节点动态扩展的需求。

* 动态调整服务节点数量，服务自动注册/删除，不用修改配置文件。
* 负载均衡在客户端实现，流量不经节点。
* 节点down掉，保证服务可用。



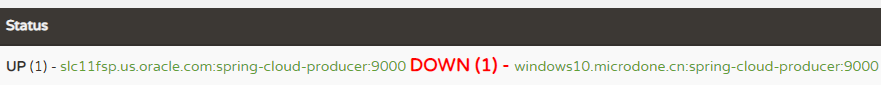
可用性

保证AP，withOutC，更适用注册发现场景。因为注册服务更重要的是可用性，可以接受短期内达不到一致性的状况。

Zookeeper: 场景：分布式协调,保证CP，withoutA, zookeeper节点点数据是同步的，极端环境可能丢弃一些请求保证一致性(client需要重新请求)。例如：当master失联时，剩余节点选举期间整个zk集群都是不可用的。

自我保护模式

节点短时间丢失大量心跳连接(可能发生了网络故障)，这个节点会进入“自我保护模式”，同时保留那些“心跳死亡”注册信息。以防还有客户端向其发起请求。故障恢复，这个节点会退出此模式。Eureka的哲学是，同时保留“好数据”与“坏数据”总比丢掉任何数据要更好。



集群

zookeeper和eureka集群都是一种去中心化架构，每个节点保存完整实例注册服务信息, 是平等的。

原理

Eureka Server：在运行过程中，Server之间会定时同步实例的注册信息。

Service Consumer：启动时，从Server获取服务信息并定时更新。

#### Feign

基于http的rpc服务调用框架，简化服务调用。整合了Ribbon，Hystrix和RestTemplate，。

配置

可直接用配置Ribbon的方式来自定义调用的参数。负载均衡策略

* RoundRobin：轮询
* Random
* WeightedResponseTime
* ZoneAware

feign声明式Web客户端

1. @FeignClient(name= "spring-cloud-producer",fallback = HelloHystrix.**class**)
2. **public** **interface** HelloRemote {
3. @RequestMapping(value = "/hello")
4. **public** String hello(@RequestParam(value = "name") String name);
5. }

Request Interceptor

1. @Configuration
2. **public** **class** HmilyFeignInterceptor **implements** RequestInterceptor {
3. @Override
4. **public** **void** apply(**final** RequestTemplate requestTemplate) {
5. Context Context = ContextLocal.get();
6. requestTemplate.header("context", GsonUtils.toJson(context));
7. }
8. }

Todo

Spring Cloud Loadbalance

#### Spring Cloud Config

随着项目规模变大，每个项目都散落着各种配置文件，如果采用分布式的开发模式，需要的配置文件随着服务增加而不断增多。某一个基础服务信息变更，都会引起一系列的更新和重启。

* 集中管理各环境的配置文件
* 配置文件修改之后，可以快速的生效。（通知）
* 可以进行版本管理

springboot项目只有在启动的时候才会获取配置文件的值，更新时需手动refresh

Spring Cloud Bus

本质是利用了MQ的广播机制在分布式的系统中传播消息，目前常用的有Kafka和RabbitMQ。

#### Hystrix

熔断器，用于服务容错，处理服务的熔断防止故障扩散。

[Reference](http://www.ityouknow.com/springcloud/2017/05/16/springcloud-hystrix.html)

如果它在一段时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程服务器，从而防止应用程序不断地尝试执行可能会失败的操作，使得应用程序继续执行而不用等待修正错误，或者浪费CPU时间去等到长时间的超时产生。熔断器也可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试调用操作。

熔断器模式就像是那些容易导致错误的操作的一种代理。这种代理能够记录最近调用发生错误的次数，然后决定使用允许操作继续，或者立即返回错误。

#### Gateway

Gateway将外部request均衡分发给后台服务端。

* 简化调用：后端实例动态变化，即服务的访问地址。网关简化外部客户端调用。
* 安全：在网关认证，无需在每个微服务API认证。

在网关层实现非业务逻辑，比如监控，提高业务灵活性。此外在网关中实现 日志/限流等

注册中心 VS Gateway

* 注册中心：构建内部服务，。
* Gateway用于服务外部调用者。

Nginx API Gateway

API Gateway诞生为解决云端问题，而Nginx仅作为反向代理，没有上面的功能。

##### Spring Cloud Zuul

简单使用

1. zuul.routes.hello.path=/hello/\*\*
2. zuul.routes.hello.url=http://localhost:9000/

通过url映射的方式来实现zull的转发有局限性，比如每增加一个服务就需要配置一条内容，另外后端的服务如果是动态来提供，就不能采用这种方案来配置了。

服务化

服务名与服务实例地址的关系在eureka中已存在，将Zuul注册到eureka即可发现其他服务。

1. zuul.routes.api-a.path=/producer/\*\*
2. zuul.routes.api-a.serviceId=spring-cloud-producer

serviceId是服务名，zuul默认路由规则如下：<http://zuul_host:zuul_port/serviceId/path>,

比如 http://localhost:8888/spring-cloud-producer/hello?name=%E5%B0%8F

路由熔断

后端服务出现异常的时候，我们不希望将异常抛出给最外层。

断路器就只有在该服务的所有实例都无法运作的情况下才能起作用。这种时候，断路器的形式更像是提供一种友好的错误信息，或者假装服务正常运行的假象给使用者。

##### Spring Cloud Gateway

Zuul（1.x） 基于 Servlet，使用阻塞 API，它不支持任何长连接，如 WebSockets，Spring Cloud Gateway 使用非阻塞 API，支持 WebSockets，支持限流等新特性。

设计准则

服务大小

不是越小越好，

设计原则

* 单一职责
* 保证微服务设计能支持服务的敏捷/独立地开发和部署；

微服务划分的粒度？**M**

数据管理

每个服务有自己的DB,去中心化的数据管理

服务治理

去中心化的治理，服务发现可以在API网关级别实现。

跨服务调用

#### ELK日志处理

请求加traceID记录日志。集群环境也会遇到，

ELK工具

Elasticsearch：分布式搜索和分析引擎。全文检索，结构化和分析。

Logstash：实时传输数据的管道。可🏠过滤等

Kibana ：es的展示界面，展示数据，分析数据

Zipkin捕捉应用中未捕捉的异常，加了Hystrix看不出异常，

#### Sleuth+Dapper/zipkin

分布式服务追踪。主要三部分 数据收集🡪存储🡪展示

#### Spring Boot Admin

服务监控，监控内容。

Spring Boot Actuator 提供了对单个 Spring Boot 的监控，信息包含：应用状态、内存、线程、堆栈等等，比较全面的监控了 Spring Boot 应用的整个生命周期。

# 第三篇 分布式系统

## 什么是分布式架构

多个节点组成的系统，节点通过消息传递协同工作。

需求：互联网的兴起，大型网站的各种高并发访问、海量数据处理等场景越来越多。(突破单机上线)

目标：实现网站的高可用、易伸缩、可扩展、安全等。

### 3.1.1 大型网站架构演化

大型网站：高并发+海量数据+业务复杂



#### 单体架构

当网站流量很小时，只要一个应用，所有功能都部署在一台机器，方便维护，成本又低。访问量增加，可做集群。

特点：一般采用昂贵的大型主机,结构简单，不用考虑各节点之间的分布式协作问题。

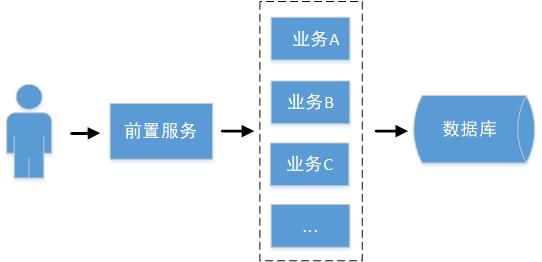
问题：单体架构发展大一定程度，项目复杂度高，难于维护、扩展性差，影响开发效率等。

实例理解

企业软件设计需要满足多种多样的业务需求。因此，一个特定的应用软件会包括有几百个功能项，而所有这些功能项都打包进了一个单体的应用中。典型的例子有，ERP、CRM等其他各种各样的软件。对于这种野兽级别的软件应用、部署、排错、扩展和升级工作都是一个个噩梦。

#### 垂直架构

又叫业务拆分。将复杂的业务拆分成不同的产品线，分而治之，以降低复杂度，提升效率。



如图垂直架构最多的还是通过同一DB整合业务。

* 拆分：拆分后的应用独立部署维护。
* 整合：超链接/消息队列/同一个DB。或者这几种方式的组合。

问题

随着业务拆分越来越小，复杂度呈指数级增加，部署维护越来越困难。

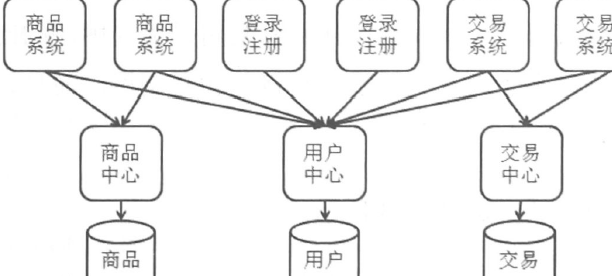
1）跨应用的交叉调用越来越复杂

2）db连接数目是服务器规模的平方。导致db资源不足。

#### 分布式架构

抽取核心业务，作为独立服务，使用独立的DB。

大型网站架构演化到这里，大多数技术问题得以解决。因此，目前一些云计算平台，将计算作为基础资源出售，中小网站按需付费。



架构的本质

从单体架构到服务化架构，应用数量都在不断的增加，基础组建下沉，业务系统上浮。

变化的原因:用户规模，业务带来的复杂度变化（并发量/数据量/业务复杂度）。

目前通过引用jar包，耦合度较高。

#### SOA

面向服务的架构：就像面向对象的程序由一个一个类组成。1）SOA的软件可以理解为由一个一个服务组成。2）每个服务独立发布独立运行）。

问题：服务越来越多，交互越发复杂，不可避免出现资源浪费（计算机资源）的情况。

目的：服务治理，提高集群利用率。

方法：增加调用中心，基于访问压力实时管理集群容量。

在这个阶段可以使用WebService或者dubbo来服务治理。

#### 微服务

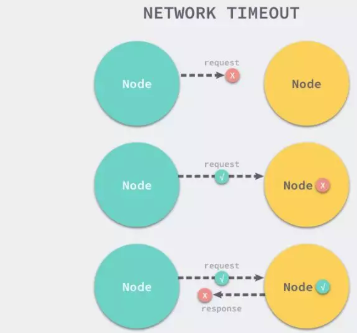
微服务架构是一种架构模式，强调彻底的服务化（拆分粒度更小）和组件化

### 3.2.1 网络请求

网络通信经常是不稳定的，网络通信可能会包含成功、失败以及超时三种情况。

在分布式架构中，服务之间基于网络通信，所以会出现障碍。

由此带来的问题



比如网络延迟，重复处理，消息丢失等问题。

**重复消费**

下游业务接口保证幂等。

Kafka提供exactly once但consumer的下游必须是kafka。

**消息丢失**

上游：重试

网络请求超时是造成分布式复杂的根本原因。

* 可靠性：节点通过网络来交换信息，网络超时引发一系列问题。
* 复杂度：由于可靠性问题，增加系统处理的复杂度（比如分布式事务）。
* 单点故障：一般采用普通计算机，服务器数量多，因此宕机概率大。

分布式 VS 集群

分布式：一个业务分拆多出个子业务，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

通过接口分解系统解耦性，不同子系统通过相同的接口描述进行服务调用。

### 3.2.3 数据治理

SpringBoot+MongoDB

Todo

3.4 SOA服务治理

服务治理

1）注册中心：管理所有服务

2）路由选择、负载均衡及容错处理；

3）服务监控与统计；

4）服务过滤（黑名单、白名单）；

5）服务升降级，权重调整；

6）服务状态检测、监测；

7）服务权限控制。

8）服务依赖关系；

注册中心

[Reference](https://mp.weixin.qq.com/s?__biz=MzIyMzgyODkxMQ==&mid=2247483885&idx=1&sn=73321a5574045c47fccb3013e47ec0e9&chksm=e8190f2ddf6e863b6016bd90b396cf9eaa2365ff42d4120299de2ec5b4f510ad071ce1d079f8&scene=21#wechat_redirect)

需求：1）根据负载，动态调整服务节点数量。2）节点down掉，保证服务可用。

方案一 Redis：

1) 频繁轮询

2) 缓存解决轮询，带来滞后性问题

3) 节点新增或者删除（down掉或kill）服务节点，无法及时清除

方案二 zookeeper

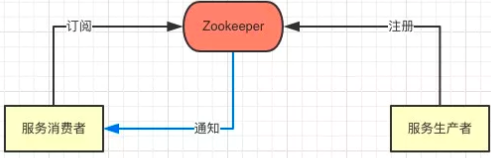
Zookeeper

Zookeeper是一个分布式一致性框架。主要用来解决分布式集群中应用系统的一致性问题。

本质：缓存+推送

观察者模式

缓存数据+订阅发布



服务生产者删除时，zookeeper通知消费者更新本地缓存并删除保存的节点信息。

使用场景

配置中心：统一的配置中心，不需要在每个服务器单独配置（通知模式）。

分布式锁：瞬时有序节点

TODO

机器间存在依赖关系时，就需要处理协调问题。如指定一台作为master主节点接收任务，并将任务分为其他从节点，涉及到主节点选举和主节点是否存活的判断。另外这些机器作为RPC服务提供者提供服务时，需要提供服务注册查询功能。当我们想实现一个分布式锁时。这些都是Zookeeper可以发挥的场景。总结来说，

* 服务发现
* 分布式锁
* 分布式应用可以基于它实现同步，配置管理，集群管理，命名名空间
* 崩溃检查
* 存储分布式元数据
* 选举主节点

节点类型**:**永久节点;瞬时节点

是否可以做分布式的数据库？

https://www.zhihu.com/question/22116083

CMD

/zkServer.sh start/status

路由策略

考虑重点：1.尽量保证各节点的连接均匀。2.增删节点是否要做 Rebalance。

均衡算法

* 轮询：会出现新增节点分配不均。
* Hash取模：类似HashMap，也会出现轮询的问题。可做rebalance，索引客户端重连。
* 权重：基于监控，根据负载调整，高就调低权重，低就提高权重。

推送路由

服务监控

* 各节点OS，
* JVM,GC等
* 业务监控：比如消息系统的在线数。

## 部署平台

Docker

容器即应用，面向容器开发

如你所知，为了加快项目进度，你有一个待办事项清单：

* 拉取代码基线
* 安装外部工具，如数据库、缓存和一些额外的工具和服务
* 给外部工具打补丁和升级
* 配置数据库和服务以使他们可以通信
* 开始祈祷（可能需要好几次）
* 调试问题（至少要30几次）

想象一下，如果在将应用程序的代码基线拉出存储库之后，您只需运行几个命令行（可能只有一个），可以使整个应用程序的环境准备就绪。听起来很酷，对吧？

DevOps

[Reference](https://zh.wikipedia.org/wiki/DevOps)

* 更小、更频繁的变更──意味着更少的风险
* 让开发人员更多地控制生产环境
* 更多地以应用程序为中心来理解基础设施
* 定义简洁明了的流程
* 尽可能地自动化
* 促成开发与运营的协作

### 3.7.3 后台服务

后台服务：主要包括消息系统，分布式缓存，分布式数据访问层和任务调度系统。（中间件？）

分布式数据访问层：分表分库

## 3.3 消息队列

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMTY0NDU3Nw==&mid=2651938493&idx=1&sn=d0a18b9dd5d587726196310df1811d8d&chksm=8d0f31f3ba78b8e5541ea73b3d093db52b92ff3e190bb7f594f03c9b16778db00a4e15f3b52a&mpshare=1&scene=1&srcid=#rd)

消息队列是分布式系统中重要的组件。

### 使用场景

* 系统集成：分布式系统集成方式除了直接服务调用还可以用消息队列。
* 服务解耦：基于订阅发布，需要消息的系统直接订阅。比如下单通知，耦合每个通知对象。
* 异步：非必要业务异步执行，加快响应速度。比如下单成功，发短信，发红包。
* 削峰限流：请求加入消息队列，将处理的请求控制到server能处理的范围内，比如秒杀系统类型。比Redis实现的功能强大。

MQ VS Queue

JDK Queue都是简单的内存队列。MQ 用于分布式环境（思考hashmap和redis）。使用MQ增加系统复杂度，需要考虑可用行，一致性，传输可靠性，消息重复消费等问题。

MQ产品

* ActiveMQ：同RabbitMQ老牌的。份额较少，下一代改名Apollo。
* RabbitMQ：发布频率，社区活跃，适合中小型软件公司，简单，性能佳。
* Rocketmq：阿里开源，适合数据量大，高并发场景，适合大型软件公司，有足够资金搭建分布式环境，有定制化开发能力和精力。
* kafka：同样适合分布式环境。目前最火
* ZeroMQ：轻量级消息系统。
* redis：

集群

RabbitMQ: [Reference](https://www.jianshu.com/p/79ca08116d57)

### Kafka

消息投递语义

出现网络请求超时问题，消息发送者有两种选择

重试：消息的发送者通过重试对消息进行重发，可能会导致消息的重复发送与处理

不重试：可能导致消息的丢失

* 最多一次（At-Most Once）
* 最少一次（At-Least Once）
* 正好一次（Exactly Once）

分布式系统与消息投递

[Reference](https://mp.weixin.qq.com/s?__biz=MzU0OTE4MzYzMw==&mid=2247485980&idx=1&sn=a33f504b4ef6265c4ec9c2d18eaa5380&chksm=fbb283e2ccc50af415b1a4bdafd3e26b9d284c02194cc2d5d4414b93d6aa3acccd49851a0e7f&mpshare=1&scene=1&srcid=&pass_ticket=rMfcWI31qoBc%2BJleH2)

重复消费

可靠性传输

丢失消息：

Producer

MQ

Consumer

实现可靠性消息服务，保证最终一致性。[Reference](https://my.oschina.net/xliangbo/blog/1545040)

消息顺序性

消息顺序放同一队列，只用一个consumer去消费队列

Kafka

日志处理

ActiveMQ

基于JMS

## 3.5 分布式事务

从广义上来看，分布式事务其实也是事务，比如：垂直拆分Product Order User到三个库。

刚性事务

* 2PC：强一致性。阻塞提交，而且对性能影响太大；

柔性事务

一般在应用层实现，最终一致性。

* TCC：缺点要写很多补偿的代码，开发成本高。
* 本地消息表：消息表会耦合到业务系统，不太优雅；
* MQ事务：依赖于有事务消息的MQ中间件。同本地消息表不适合下游业务需要回滚场景。
* Saga：

Reference

[关于分布式事务、两阶段提交协议、三阶提交协议](https://www.hollischuang.com/archives/2591)

[柔性事务与服务模式](https://www.hollischuang.com/archives/2591)

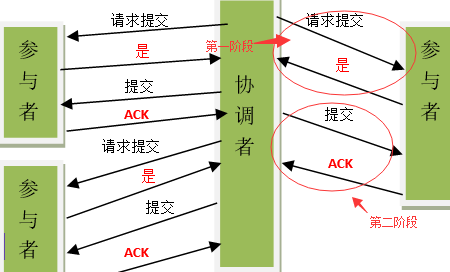
除了XA ，其他方案都会侵入业务层，需要设计实现正向和反向的幂等接口，往往会导致很高的研发和维护成本。

### 3.4.3 2PC

二阶提交协议，主要保证了分布式事务的原子性。

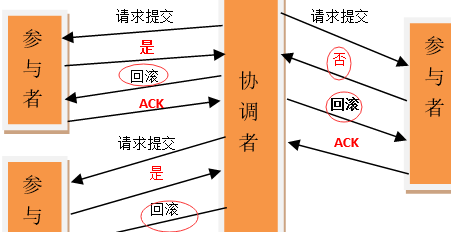
第一阶段：参与者收到协调者请求提交后即提交本地事务，但没commit

第二阶段：各节点执行commit



参与者：RM，协调者：TM

任何节点的响应消息为“中止“或询问超时，所有节点在第二阶段结束事务。



几个问题：

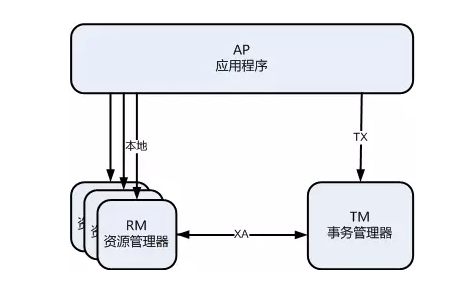
单点故障：协调者故障，导致所有参与者一直阻塞。

数据不一致：协调者故障（第二阶段），只有部分参与者收到提交请求并执行。

XA规范

XA是分布式事务处理（DTP）的标准协议，保证分布式事务的ACID特性，像本地事务一样。

DTP模型



* TM : Transaction Manager。
* RM : Resource Manager，由DB实现，比如Oracle、DB2都实现了XA接口
* XA 协议描述了 TM 与 RM 之间的接口

CRM ：通信资源管理器，比如消息中间件。

AP：Application Program。

二阶提交协议和三阶提交协议是XA的具体实现。

### 3.4.4 TCC

采用补偿机制，对每个操作注册一个与对应的确认和补偿（撤销）操作。

* try：业务校验和资源预留（DML）。
* confirm：确认执行业务操作（Commit）。
* cancel：取消执行业务操作（Rollback）。

1.为什么要try不直接一步完成？

单体数量多出错概率高。

2.为什么需要要资源预留？

2PC的阻塞并不适用高并发场景，TCC在应用层实现，单体资源预留即可提交，实现最终一致性。

case：同步调用

实现步骤

1. **public** Transaction interceptor(ProceedingJoinPoint point) {
2. Transaction trans=preTry();//1.buid trans
3. **try**{
4. returnValue = point.proceed();////2.execute try
5. }**catch**{
6. cancel(trans);
7. }
8. confirm(trans); //confirm
9. }

Aop

1. **public** Transaction preTry() {
2. Transaction trans = buildTransaction(point);//PRE\_TRY
3. saveTransaction(trans);
4. **return** trans;
5. }

TODO

[分布式事务的实现原理](https://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247485028&idx=1&sn=6473cc9f4854ebb689449e62391c337a&chksm=9bed2716ac9aae00e438a1221d22e1f0ba4f6ffe2f9808d1384d50eba8d9f02217a1c420e03c&mpshare=1&scene=1&srcid=#rd])

[增加一个协调服务](https://www.jianshu.com/p/2d2867349f66)

### 3.4.5 MQ事务/可靠消息服务

单体的事务直接提交🡪mq🡪所有的参与者

* 没有补偿发生异常将日志记录，admin查看
* 发起者service里做好业务校验，尽量确保全局事物成功提交。
* 每个单体都要有张事务表。

全局事务

角色：发起者和参与者

事务状态：开始,已提交,失败

Producter

1. **try**{
2. begin:tranId=save to tran\_table
3. exeService(commit)
4. updateTranTable(COMMIT)
5. }**catch**{
6. updateTranTable(FAILED)
7. }**finally**{
8. pushToMQ(tranId)//  只要调用过的下游方法且需要纳入同一tran的都要push
9. }

Schedule Check: 查询tran\_table中status=begin的,防止在2之后down机。

**Consumer**

push/pull from MQ

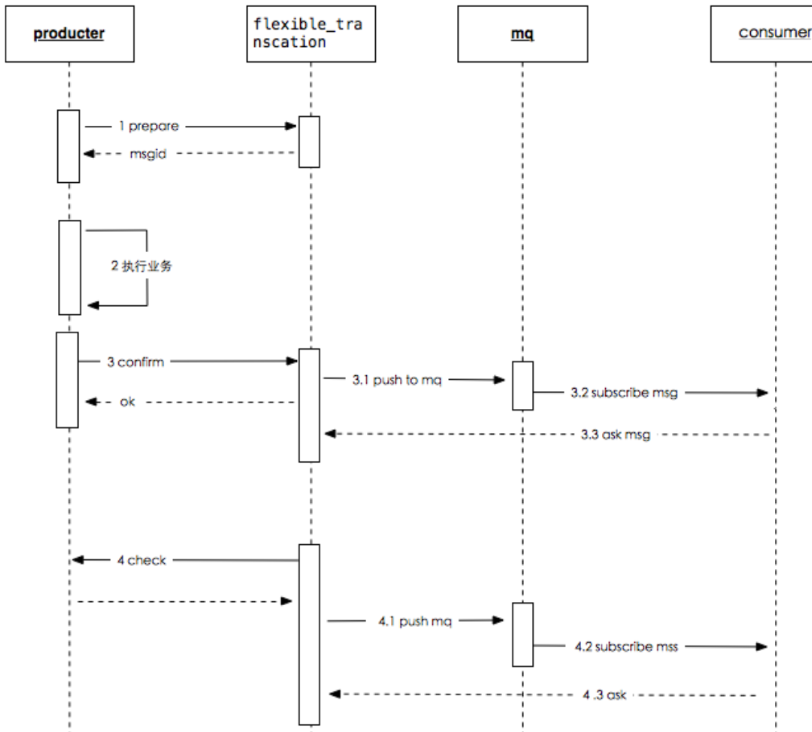
缺点：

* 实现难度大，主流MQ不支持。
* 下游业务需要回滚需要tcc补偿机制（同本地消息表）。

校验提到发起者

能确保事务消息发送成功且一定被消费。主流MQ不支持事务消息，比如Kafka和RabbitMQ。

[Reference](https://github.com/smartxing/flexible-transcation)



大致的思路：

每个单体应用都create tran\_table。对应用没有侵入。

本地消息表



优点： 一种非常经典的实现，避免了分布式事务，实现了最终一致性。

缺点： 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

Paxos算法

Chubby是google的paxos算法的实现，用来实现分布式锁、Master选举等功能。zookeeper是Chubby的开源实现，在Hadoop、Storm、Kafka等众多开源软件中都使用了zookeeper，各大互联网公司也都使用zookeeper作为分布式协调服务。

Saga

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247485028&idx=1&sn=6473cc9f4854ebb689449e62391c337a&chksm=9bed2716ac9aae00e438a1221d22e1f0ba4f6ffe2f9808d1384d50eba8d9f02217a1c420e03c&mpshare=1&scene=1&srcid=#rd])

todo

JTA事务

Java事务的类型：JDBC事务、JTA(Java Transaction API)事务、容器事务。 常见的容器事务如Spring事务，容器事务主要是J2EE应用服务器提供的，容器事务大多是基于JTA完成，这是一个基于JNDI的，相当复杂的API实现。

JDBC事务有一个局限：一个 JDBC 事务不能跨越多个数据库.

### 3.4.1 理论支持

#### CAP定理

[Reference](https://www.hollischuang.com/archives/666)

一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

分区容错性可理解为分区

Warning

1. CAP的前提是P，有P才有CA的抉择，舍弃等同于单机系统。
2. 这里的一致性是强一致性。下面讲得牺牲一致性指强一致性，但可到达最终一致性。

CAP

1）Consistency

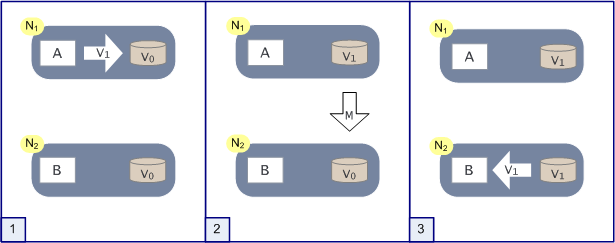
all nodes see the same data at the same time。强一致性。

2）Availability

Reads and writes always succeed，即服务一直可用，而且是正常响应时间。

3）Partition Tolerance

分区：指分布式，分成N个节点。分区容错性指分布式系统在遇到某节点或网络分区故障的时，单个节点仍能对外提供满足CA服务。



节点：N1,N2

一致性: N1和N2的数据应一样。V0=V0 。

可用性：请求N1和N2应得到立即响应。

分区容错性：N1和N2任何一方故障，彼此正常运转。

CAP权衡

P是分布式的基础，舍弃等同于单机系统。提升基础设施的稳定性才能保障P。

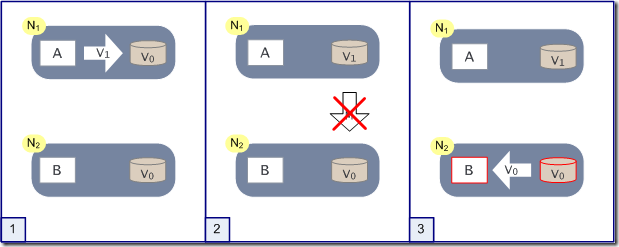
Without A

舍弃可用性。在分布式系统要想获得强一致性就要牺牲可用性。

典型的就是很多分布式数据库，在发生极端情况时，优先保证数据的强一致性，代价就是舍弃系统的可用性。如Redis、HBase等，还有分布式系统中常用的Zookeeper也是在CAP三者之中选择优先保证CP的。

Without C

舍弃一致性。



使用场景

在12306买票，查看时有票的，下单提示余票不足。牺牲一致性会影响一些用户体验，但是不至于造成流程的严重阻塞。准确的说这里舍弃的只是强一致性。退而求其次保证了最终一致性。

#### BASE理论

Base理论是对CAP理论的延伸。核心思想：即使无法做到强一致性，应用可以采用适合的方式达到最终一致性。

1）Basically Available

基本可用是指分布式系统在出现故障的时候，允许损失部分可用性，保证核心可用。

电商大促时，为了应对访问量激增，部分用户可能会被引导到降级页面，服务层也可能只提供降级服务。这就是损失部分可用性的体现。

2）Soft State

软状态是指允许系统存在中间状态。

分布式存储中一般一份数据至少会有三个副本，允许不同节点间副本同步的延时就是软状态的体现。MySQL replication的异步复制也是一种体现。

3）Eventual Consistency

最终一致性是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。弱一致性和强一致性相反，最终一致性是弱一致性的一种特殊情况。

ACID VS BASE

ACID是传统数据库设计理论，追求强一致性。这里的C指数据的和数据关系的正确性。

BASE支持的是大型分布式系统，提出牺牲强一致性获得高可用性。