

ccc

前 言

往事越千年，魏武挥鞭，东临碣石有遗篇。萧瑟秋风今又是，换了人间。



The Sea

浙江 杭州



扫一扫上面的二维码图案，加我微信

微信

目 录

概 览

为什么考研究僧，每个参与考试的人都有自己的想法。

相信，付出努力，就会有收获。

相信，当你觉得最迟的时候，恰恰是最早的时候。

相信，当你足够努力的时候，全世界都会为你让路。

相信，读书改变命运。

杭电，有人均两篇SCI的云计算实验室。

杭电，有人北大硕士MIT博士的顶级助教。

杭电，有数百块泰坦GPU构建的深度学习集群。

杭电，有对世界的抽象-图形图像实验室。

.....

本人以题典的形式将杭电的题型抽象出来。每道题都是对知识点的串接。学懂了每道题，也就对每个重要知识点的掌握。

1

题典之Hash

- ▶ 知识点：讲解相关知识点。
- ▶ 题型：直接上真题。

1.1 知识点和方法论

1.1.1 知识点

- ▶ 开放定址法： $H(key)$ 为题目选定的散列函数， m 列表长度， di 为增量序列， Hi 新的位置
 - 核心公式：

$$Hi = (H(key) + di) \% m$$

- 线性探测法：

$$di = 0, 1, 2, 3, \dots, m - 1$$

- 平方探测法(又称二次探测)：

$$di = 0, 1, -1, 4, -4, \dots, k^2, -k^2 (k \leq m/2)$$

- ▶ 平均查找长度

-

$$ASL_{\text{成功}} = (\text{查找成功的次数, 第一次也算一次}) / \text{元素的个数}$$

- ▶ 平均失败查找长度

- $ASL_{\text{失败}} = (\text{在mod数范围内的空间才算}) / \text{MOD后面的数}$
- ▶ 装填因子: 衡量冲突的概率
 - $\alpha = n(\text{关键字个数}) / N(\text{表长})$
- ▶ 链地址法
 - 就像领接表那样

1.1.2 方法论

1. 画出数组
2. 后面填入数字(比较次数)

1.2 真题实战

1.2.1 2017年第8题

算法 1-1 2017年第8题

已知函数为
 $\text{HashH}(K) = K \bmod 13$ ，散列地址为，用开放地址法解决冲突，选取增量序列为线性探测再散列，关键字0--14
 23, 34, 56, 24, 75, 12, 59, 52, 36, 92 依次插入到散列表中，则平均成功的查找长度为_____、平均失败的查找长度为_____。

解：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
52(1)	36(7)	92(2)		56(1)				34(1)		23(1)	24(1)	75(3)	12(2)	49(5)
4	3	2	1	2	1	1	1	2	1	9	8	7	6	5

23 % 13 = 10
34 % 13 = 8
56 % 13 = 4
24 % 13 = 11
75 % 13 = 10 冲突 (10 + 1) % 15 = 11 冲突 (10 +)
2% 15 = 12
12 % 13 = 12 冲突 (12 + 1) % 15 = 13
49 % 13 = 10 冲突 (10 + 1) % 15 = 11 冲突 (10 +)
2% 15 = 12 冲突 (10 + 3) % 15 = 13 冲突 (10 + 4) % 15 = 14

$52 \% 13 = 0$ $36 \% 13 = 10$ 冲突 $(10 + 1) \% 15 = 11$ 冲突 $(10 +)$ $2 \% 15 = 12$ 冲突 $(10 + 3) \% 15 = 13$ 冲 $突 (10 + 4) \% 15 = 14$ $(10 +) 5 \% 15 = 0$ 冲突 $(10 +) 6 \% 15 = 1$ $92 \% 13 = 1$ $() 1 + 1 \% 15 = 2$

$$1 + 7 + 2 + 1 + 1 + 1 + 3 + 2 + 5 = 24 \text{ (查找次数)} \quad (1.1)$$

$$24 / 10 = 2.4 \text{ (平均查找长度)} \quad (1.2)$$

对于0地址的元素要查找0,1,2,3这几个元素才知道会不会失败，第三个是空元素，所以失败了对于1地址的元素要查找1,2,3这几个元素才知道会不会失败，第3个元素是空元素，所以失败了以此类推因为 mod 13 只用看 0 - 12 空间里面的错误

$$(4 + 3 + 2 + 1 + 2 + 1 + 1 + 1 + 2 + 1 + 9 + 8 + 7) = 42 \quad (1.3)$$

$$ASL_{\text{失败}} = 42 / 13 \quad (1.4)$$

1.2.2 王道269综合应用题第3题

3. 使用散列函数 $H(key) = key \% 11$, 把一个整数值转换成散列表下标, 现要把数据 {1, 13, 12, 34, 38, 33, 27, 22} 依次插入到散列表中。
 1) 使用线性探测法来构造散列表。
 2) 使用链地址法构造散列表。
 试针对这两种情况, 分别确定查找成功所需的平均查找长度, 及查找不成功所需的平均查找长度。



图 1-1 王道269综合应用题第3题

解:

1)

0	1	2	3	4	5	6	7	8	9	10
33(1)	1(1)	13(1)	12(3)	34(4)	38(1)	27(2)	22(8)			

```
1 % 11 = 1
13 % 11 = 2
12 % 11 = 冲突1    () 1+1% 10 = 2 冲突    (1 + 2) % 10 = 3
34 % 11 = 冲突1    () 1+1% 10 = 2 冲突    (1 + 2) % 10 = 3 冲
突(1+ 3) % 10 = 4
33 % 11 = 0
38 % 11 = 5
27 % 11 = 5 冲突 (5+1)%10 = 6
22 % 11 = 0 冲突 1 冲突 2 冲突3 冲突4 冲突5 冲突6 冲突7
```

2)

拉链法只要算一次

```
1 % 11 = 1
13 % 11 = 2
12 % 11 = 1
34 % 11 = 1
33 % 11 = 0
38 % 11 = 5
27 % 11 = 5
22 % 11 = 0
```

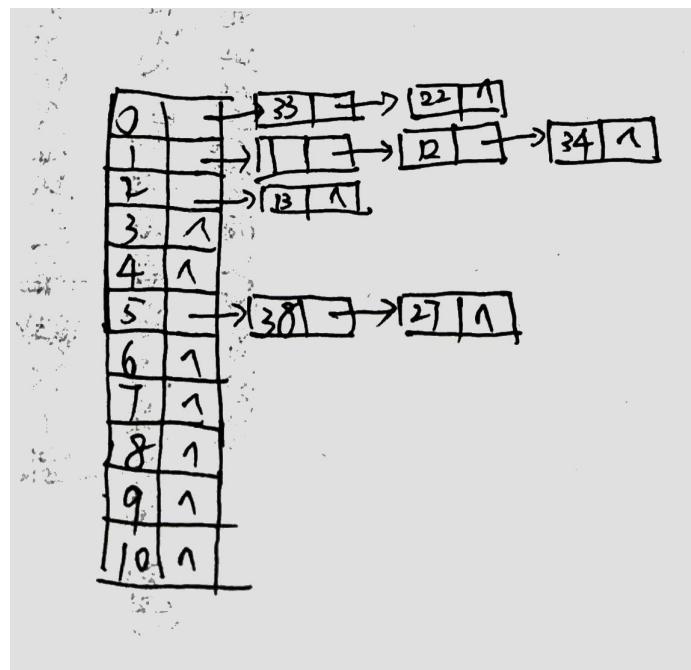


图 1-2 王道269综合应用题第3题

3)

用

A B 指代 1) 2)

A

$$A(\text{ASL 成功}) = (1 + 1 + 1 + 3 + 4 + 1 + 2 + 8) / 8 = 21 / 8$$

$$A(\text{ASL 失败}) = (9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 1 + 1) / 11 = 47 / 11$$

B

$$B(\text{成功ASL}) = (4 * 1 + 3 * 2 + 3 * 1) / 8 = 13 / 8$$

$$B(\text{失败ASL}) = (3 + 4 + 2 + 1 + 1 + 3 + 1 * 5) / 11 = 19 / 11$$

1.2.3 2015年第(5)题

设散列函数

$$H(K) = 3K \bmod 11, \text{ 散列地址空间为 } 0 - 10, \text{ 对关键字序}$$

列 (32, 13, 39, 24, 38, 21, 4, 12) 按照下述两种解决冲突的方法构
造散列表；

1) 线性探测再散列；

2) 链地址法；

| 3) 并分别求出等概率下查找成功时和查找失败时的平均查找长度 $ASL_{\{succ\}}$ 和 $ASL_{\{UNSUCC\}}$ }

解：

0	1	2	3	4	5	6	7	8	9	10
	4(1)		49(1)	38(1)	12(3)	13(1)	24(2)	32(1)	21(2)	

(3*32) % 11 = 8
 (13*3) % 11 = 6
 (49*3) % 11 = 3
 (24*3) % 11 = 6 冲突 (6+1) % 11 = 7
 (38*3) % 11 = 4
 (21*3) % 11 = 8 冲突 (8+1) % 11 = 9
 (4*3) % 11 = 1
 (12*3) % 11 = 3 冲突 4 冲突 5

$$ASL_{\text{失败}} = (1 + 2 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1) / 11 = 39 / 11 \quad (1.5)$$

$$ASL_{\text{成功}} = (1 + 1 + 1 + 3 + 1 + 2 + 1 + 2) / 8 = 1.5 \quad (1.6)$$

$$ASL_{\text{失败}} = (1 + 2 + 1 + 3 + 2 + 1 + 3 + 1 + 3 + 1 + 1) / 11 = 19 / 11 \quad (1.7)$$

$$ASL_{\text{成功}} = (5 * 1 + 3 * 2) / 8 = 11 / 8 \quad (1.8)$$

1.2.4 2013年第4题

设哈希函数为

$H(key) = key \bmod 13$ 哈希表长为，用开放定址法处理冲突，增量序列使用
二次探测再散列。若一次在哈希表中插入个元素：1511

34, 12, 67, 43, 98, 23, 51, 86, 05, 37, 22

1) 画出他们在表中的分布情形。

2) 求其等概率情况下平均成功的查找长度

解：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		67(1)	22(6)	43(1)	5(1)		98(1)	34(1)	86(2)	23(1)	37(1)	12(1)	51(2)	

```

39 % 13 = 8
12 % 13 = 12
67 % 13 = 2
43 % 13 = 4
98 % 13 = 7
23 % 13 = 10
51 % 13 = 12 冲突 (12+1) % 15 = 13
86 % 13 = 8 冲突 (8+1) % 15 = 9
5 % 13 = 5
37 % 13 = 11
22 % 13 = 9 冲突 (9+1) % 15 = 10 冲突 (9-1)%15 = 8 冲
突 (9+4) %15 = 13 冲突 (9-4) %15 = 5 冲
突 (9+9) %15 = 3

```

$$ASL_{\text{成功}} = (1 + 6 + 1 + 1 + 1 + 2 + 1 + 1 + 1 + 2) / 11 = 18/11 \quad (1.9)$$

1.2.5 2014年第4题

采用哈希后函数

$H(k) = 3*k \bmod 13$ 并用开放地址法处理冲突，增量序列选取采用线性探测再散列方式，在数列地址空间中对关键字序列[0..12]

22, 41, 53, 46, 30, 13, 1, 67, 51

- 1) 构造哈希表画示意图();
- 2) 装填因子;
- 3) 查找成功时的平均查找长度;
- 4) 查找不成功时的平均查找长度。

解：

0	1	2	3	4	5	6	7	8	9	10	11	12
13,1	66,1		53,1	1,2		41,1	67,2	46,1		51,1		30,1

```

3*22 % 13 = 1
41*3 % 13 = 6
53*3 % 13 = 3
46*3 % 13 = 8
30*3 % 13 = 12
13*3 % 13 = 0
(1*3)% 13 = 3 冲突 4
(667*3)%13 = 6 冲突 7
(51*3) %13 = 10

```

$$\alpha = \frac{n}{N} = \frac{9}{13} \quad (1.10)$$

$$ASL_{\text{成功}} = (1 + 1 + 1 + 2 + 1 + 2 + 1 + 1 + 1)/9 = 11/9 \quad (1.11)$$

$$ASL_{\text{失败}} = (3 + 2 + 1 + 3 + 2 + 1 + 4 + 3 + 2 + 1 + 2 + 1 + 4)/13 = 29/13 \quad (1.12)$$

2

计算机体系结构

► ...

2.1 知识点和方法论

2.1.1 知识点

2.1.1.1 CPU内存模型

多个cpu每个cpu有自己的cache缓存, 他们通过mesi协议(缓存一致性协议)和主存进行通信 **mesi协议** mesi协议保证了每个缓存中使用的共享变量的副本是一致的. 他的核心思想是: 当CPU写数据时候, 如果发现操作的变量是共享变量, 即在其他CPU中可存在该变量的副本, 会发出信号通知其他CPU将该变量的缓存行设置为无效状态, 当其他CPU需要读取这个变量时候, 发现自己缓存中缓存该变量的行是无效的, 那么它就会从内存重新读取.

2.1.1.2 linux 用户态和内核态

由于对硬件的操作涉及到驱动的调用. 这通常是不一样的. 整个操作系统为了屏蔽驱动不同的影响, 提供了一组系统调用的接口, 可以让我们利用整个机器的硬件资源, 而不用去完整理解设备的驱动逻辑.

比如我们要申请内存. 不同的内存的驱动可能是不同的, 有些内存条支持2666MHZ的频率, 但是通过系统调用可以屏蔽不同的驱动的操作方式, malloc 会调用brk()或者mmap()系统调用来分配内存.

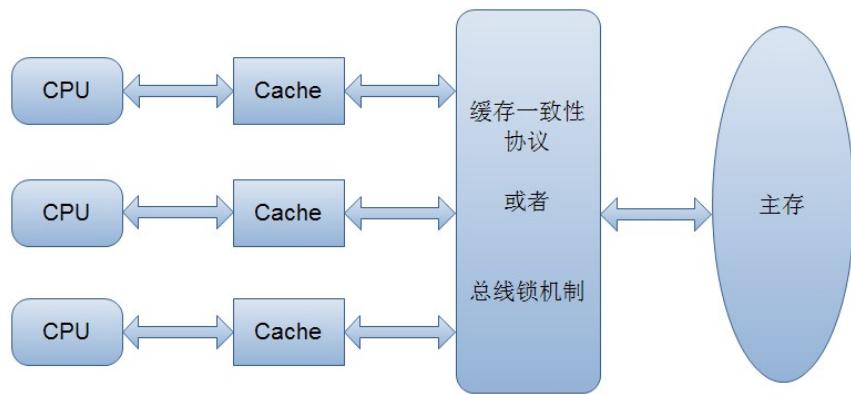


图 2-1 cpumem

从用户态到内核态切换可以通过三种方式:

1. 系统调用
2. 异常: 比如发生了缺页异常
3. 外设中断: 当外设完成用户的请求时候, 会向CPU发送中断信号.

2.1.1.3 nginx高性能原因

epoll 多路复用

master worker 进程模型, 平滑重启, master 重启, 然后将新的worker句柄给新的worker
协程机制依附于线程的内存模型, 切换开销小, 与阻塞及归还执行权, 代码同步, 无需加锁

2.1.1.4 bio select epoll

bio 如果输出端满会阻塞

select 遍历, 查询效率低

epoll 模型, 变更触发回调直接

3

Spring

► ...

3.1 知识点和方法论

3.1.1 知识点

3.1.1.1 jar 和 war 包之间的区别

jar 包集成了Tomcat

war 没有集成Tomcat

3.1.1.2 SpringBoot 和 SpringMVC 有什么区别

SpringBoot 简化了项目的开发配置流程, 一定程度上消除了xml配置, 是一套快速配置开发的脚手架.

springMvc主要解决WEB开发的问题, 是基于Servlet的一个MVC框架, 通过XML配置, 统一开发前端视图和后端逻辑;

3.1.1.3 Spring 框架能带来哪些好处

1. Dependency Injection(DI) 依赖注入是的构造器和JavaBean properties文件中的依赖关系一目了然.
2. IoC容器更加趋向于轻量级.

3.1.1.4 如何实现AOP, 项目那些地方用到了AOP

利用JDK动态代理或Cglib动态代理, 利用动态代理技术, 可以针对某个类生成代理对象, 当调用代理对象的某个方法时, 可以任意控制该方法的执行, 比如可以先打印执行时间, 再执行该方法, 并且该方法执行完成后, 再次打印执行时间.

权限管理是使用AOP技术实现的. 凡是需要对某些方法做统一处理的都可以用AOP来实现, 利用AOP可以做到业务的无侵入

3.1.1.5 Spring的事物机制

1. Spring事物机制底层是基于数据库事物和AOP机制的
2. 首先对于使用了@Transactional注解的bean, spring会创建一个代理对象作为Bean
3. 当调用代理对象的方法时, 弧线判断方法上是否加了@Transactional注解
4. 如果加了, 那么则利用事物管理器创建一个数据库连接
5. 并且修改数据库连接的autocommit属性为false, 禁止此连接的自动提交, 这是实现Spring事物非常重要的一步.
6. 然后执行当前方法, 方法中会执行sql
7. 执行完当前方法后, 如果没有出现异常就直接提交事物
8. 如果出现了异常, 并且这个异常是需要回滚的就会回滚事物, 否则仍然提交事物
9. Spring事物的隔离级别对应的就是数据库的隔离级别
10. Spring事物的传播机制是Spring事物自己实现的, 也是Spring事物中最复杂的.
11. Spring事物的传播机制是基于数据库连接来做的, 一个数据库连接一个事物, 如果传播机制配置为需要新开一个事物, 那么实际上就是先建立一个数据库连接, 在此新数据库连接上执行sql.

3.1.1.6 Spring什么时候@Transactional失效

如果某个纺纱是private的, 那么@Transactional也会失效, 因为底层cglib是基于父子类来实现的, 子类是不能重载父类的private方法的, 所以无法很好的利用代理, 也会导致@Transactional失效

3.1.1.7 介绍一下Spring, 读过源码介绍一下大致流程

1. Spring是一个快速开发框架, Spring帮助程序员来管理对象
2. Spring的源码实现的是非常优秀的, 设计模式的应用, 并发安全的实现, 面向接口的设计等

3. 在创建Spring容器, 也就是启动Spring时:
 - a. 首先会进行扫描, 骚娘得到所有的BeanDefinition对象, 并存在一个Map中
 - b. 然后筛选出非懒加载的单例BeanDefinition进行创建Bean, 对于多例Bean不需要再启动过程中去进行创建, 对于多例Bean会在每次获取Bean时利用beanDefinition去创建
 - c. 利用beanDefinition创建Bean就是Bean的创建生命周期, 这期间包括了合并BeanDefinition, 推断构造方法, 实例化, 属性填充, 初始化前, 初始化, 初始化后等步骤, 其中AOP就是发生在初始化后这一步骤中
4. 单例Bean创建完了之后, Spring会发布一个容器启动事件.
5. Spring启动结束

3.1.1.8 Autowired

使用byType和byName去寻找需要进行注入的对象

3.1.1.9 什么是控制反转(IOC)

1. 控制反转简单来说, 以前程序开发的时候, 是由程序员通过new来生成对象. 在使用控制反转的情况下, 对象的实例化由Spring框架中的IoC容器来控制对象的创建;
2. 由容器来管理这些对象的生命周期.
3. Spring中的org.springframework.beans包和org.springframework.context包构成了Spring框架Ioc的基础. 主要使用文件 applicationContext.xml 来进行配置.

3.1.1.10 什么是依赖注入?

1. Spring 通过反射来实现依赖注入
2. 当我们需要某个功能比如Connection, 至于Connection怎么构造, 何时构造我们不需要知道. 在系统运行时, Spring会在适当的时候制造一个Connection, 我们需要一个Connection, 这个Connection是由Spring注入到A中.

3.1.1.11 Spring 对对象进行创建流程

class对象反射 → 实例化 → 生成对象 → 属性填充(依赖注入) → 初始化(afterPropertiesSet) → AOP → 代理对象(cglib) → bean

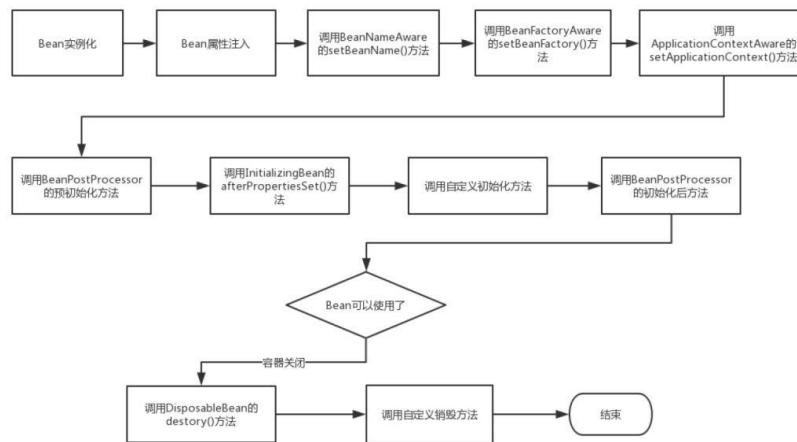


图 3-1 beanlive

3.1.1.12 什么是AOP

允许横切业务,由切面构成,切面又切入点和通知构成,@Aspect注解的类就是切面.

(1) 目标对象(Target)

要被增强的对象.

(2) 连接点,哪个目标方法,相对点,目标方法的前还是后.

3.1.1.13 bean的生命周期

什么是bean?

从上面可知,我们可以给Bean下一个定义: Bean就是由IOC实例化、组装、管理的一个对象。如上图所示, Bean 的生命周期还是比较复杂的,下面来对上图每一个步骤做文字描述:

- (1) Spring启动, 查找并加载需要被Spring管理的bean, 进行Bean的实例化
- (2) Bean实例化后对将Bean的引入和值注入到Bean的属性中
- (3) 如果Bean实现了BeanNameAware接口的话, Spring将Bean的Id传递给setBeanName()方法
- (4) 如果Bean实现了BeanFactoryAware接口的话, Spring将调用setBeanFactory()方法, 将BeanFactory容器实例传入
- (5) 如果Bean实现了ApplicationContextAware接口的话, Spring将调用Bean的setApplicationContext()方法, 将bean所在应用上下文引用传入进来。
- (6) 如果Bean实现了BeanPostProcessor接口, Spring就将调用他们的postProcessBeforeInitialization()

法。

(7) 如果Bean实现了InitializingBean接口，Spring将调用他们的afterPropertiesSet()方法。类似的，如果bean使用init-method声明了初始化方法，该方法也会被调用

(8) 如果Bean实现了BeanPostProcessor接口，Spring就将调用他们的postProcessAfterInitialization()方法。

(9) 此时，Bean已经准备就绪，可以被应用程序使用了。他们将一直驻留在应用上下文中，直到应用上下文被销毁。

(10) 如果bean实现了DisposableBean接口，Spring将调用它的destory()接口方法，同样，如果bean使用了destory-method 声明销毁方法，该方法也会被调用。

3.1.1.14 简单阐述SpringMVC的流程

SpringMVC是一个基于Java的实现了MVC设计模式的请求驱动类型的轻量级Web框架，通过把Model, View, Controller分离，将web层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发。

- (1) 用户发送请求到前端控制器DispatcherServlet;
- (2) DispatcherServlet 收到请求后，调用HandlerMapping处理器映射器，请求获取Handle
- (3) 处理器映射器根据请求url找到具体的处理器，生成处理器对象以及处理器拦截器(如果有则生成)一并返回给DispatcherServlet;
- (4) DispatcherServlet 调用HandlerAdapter处理器适配器;
- (5) HandlerAdapter 经过适配调用具体处理器(Handler, 也叫后端控制器);
- (6) Handler 执行完成返回ModelAndView;
- (7) HandlerAdapter将Handler执行结果 ModelAndView 返回给DispatcherServlet;
- (8) DispatcherServlet将 ModelAndView 传给 ViewResolver 视图解析器进行解析;
- (9) ViewResolver 解析后返回具体View
- (10) DispatcherServlet对View进行渲染视图(即将模型数据填充到视图中)
- (11) DispatcherServlet响应用户。

简单来说，我们需要开发的就是 == 开发处理器（Handler，即我们的Controller，对于视图jsp我们前后端分离之后也不用写了。

3.1.1.15 第三个三级标题

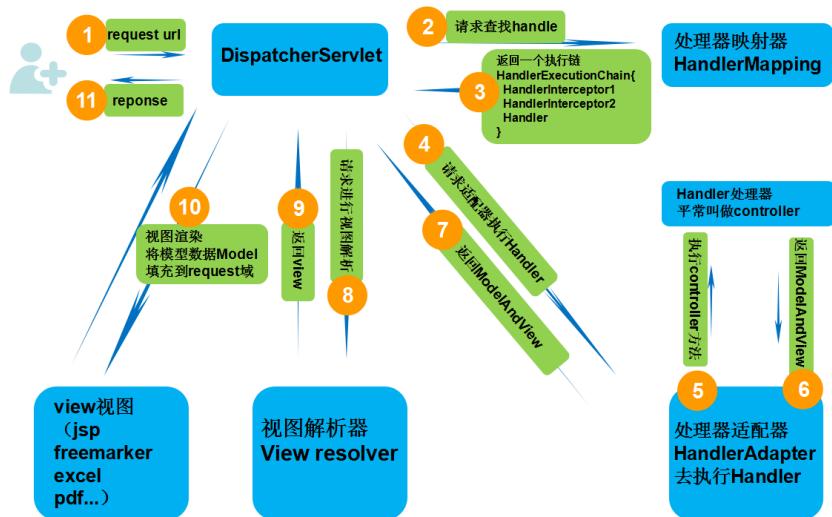


图 3-2 SpringModelAndView

4

java基础

► ...

4.1 知识点和方法论

4.1.1 知识点

4.1.1.1 双亲委派机制

Bootstrap classLoader: 主要负责加载核心的类库(java.lang.*等), 构造ExtClassLoader和APPClassLoader

ExtClassLoader: 主要负责加载jre/lib/ext目录下的一些扩展的jar。

AppClassLoader: 主要负责加载应用程序的主函数类

好处

如果有人想替换系统级别的类: String.java。篡改它的实现, 在这种机制下这些系统的类已经被Bootstrap classLoader加载过了(为什么? 因为当一个类需要加载的时候, 最先去尝试加载的就是BootstrapClassLoader), 所以其他类加载器并没有机会再去加载, 从一定程度上防止了危险代码的植入。

4.1.1.2 AQS 浅析

简单而言就是一个并发包组件, ReentrantLock 在其之上构建的.

AQS 全称: AbstractQueuedSynchronizer 抽象队列同步器

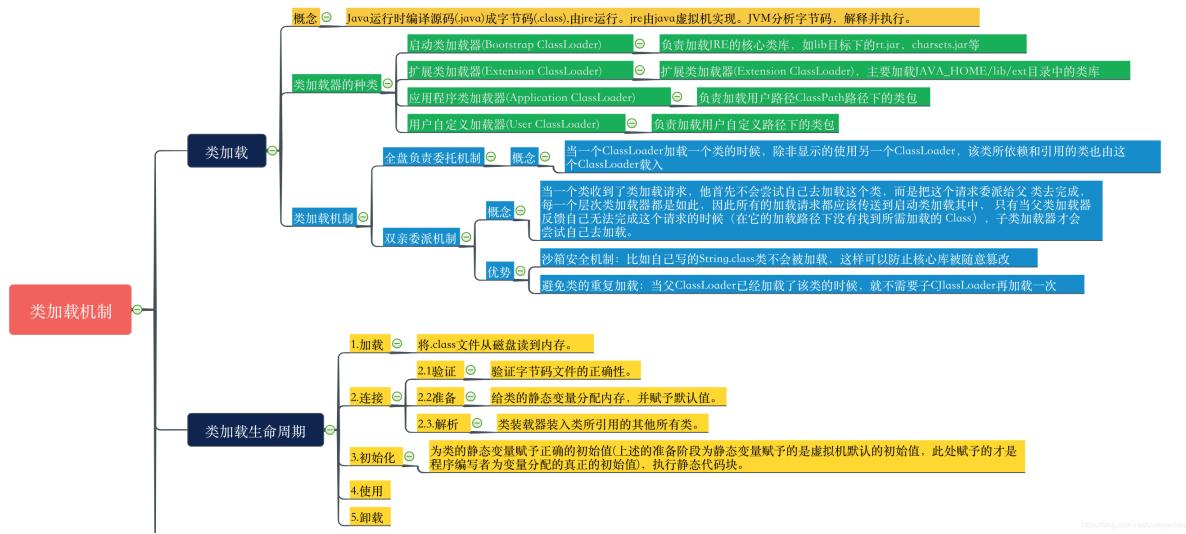


图 4-1 shuangqin

里面state=0 表示没有线程占用这个资源,还有一个对加锁线程的记录.当没有获得资源的线程就会进入等待队列,直到获取资源的线程自己释放了资源.队列中第二个线程就会被唤醒.

4.1.1.3 java加密算法浅析

MD5是不可逆加密, 不可以用来加密文本, DES和RC4是对称加密, RSA是不对称加密, 都可以用于文本加密

4.1.1.4 java 使用流的方式去除某些变量

```
List < String > ss = list.stream().filter(element -> null != element).collect(Collectors.toList());
```

4.1.1.5 java中的变量如果直接赋值为null下面直接使用的话,会报空指针异常

4.1.1.6 java接口中默认成员变量的修饰符

public static final int 静态最终变量.

public abstract 抽象方法

4.1.1.7 快速排序

参照图解<https://blog.csdn.net/pengzonglu7292/article/details/84938910>

简单的来说就是双指针交汇运动.

4.1.1.8 UML图标

<https://blog.csdn.net/u011125703/article/details/50935322>

泛华, 依赖, 关联, 聚合, 实现和组合

空心菱形是聚合

实心菱形是组合

一条直线是关联

4.1.1.9 session与token的区别?

简单来说session是以sessionid - 和 session 存储的. sessionid存储在cookie中. 每次请求会带上. token 简单来说就是一串能表示用户身份的随机字符串. 其实功能和sessionid差不多.

4.1.1.10 注解以及相关问题

注解的使用其实是一个接口继承了annotation接口, 使用@interface 来定义一个注解
一般通过反射来读取注解

有类注解, 属性注解和方法注解.

@Retention(value = RetentionPolicy.RUNTIME) 是运行时可见的注解

可以定义@Target(value= ElementType.TYPE,ElementType.METHOD,ElementType.FIELD)
是定义注解的组成

getAnnotation(MyAnnotationDefinition.class); 获取类注解

clazz.getDeclaredMethods(); 获取方法注解

Field nameField = clazz.getDeclaredField("name"); 获取属性注解

4.1.1.11 juc arraylist & int

```
static AtomicInteger atomicInteger = new AtomicInteger(0);
atomicInteger.getAndIncrement();
```

基于CAS

```
CopyOnWriteArrayList c = new CopyOnWriteArrayList();
c.add(2);
```

CopyOnWriteArrayList这是一个ArrayList的线程安全的变体，其原理大概可以通俗的理解为：初始化的时候只有一个容器，很长一段时间，这个容器数据、数量等没有发生变化的时候，大家（多个线程），都是读取（假设这段时间里只发生读取的操作）同一个容器中的数据，所以这样大家读到的数据都是唯一、一致、安全的，但是后来有人往里面增加了一个数据，这个时候CopyOnWriteArrayList 底层实现添加的原理是先copy出一个容器（可以简称副本），再往新的容器里添加这个新的数据，最后把新的容器的引用地址赋值给了之前那个旧的容器地址，但是在添加这个数据的期间，其他线程如果要去读取数据，仍然是读取到旧的容器里的数据。

使用可重入锁进行数组的锁定

```
1 public boolean add(E e) {
2     final ReentrantLock lock = this.lock;
3     lock.lock();
4     try {
5         Object[] elements = getArray();
6         int len = elements.length;
7         Object[] newElements = Arrays.copyOf(elements, len + 1);
8         newElements[len] = e;
9         setArray(newElements);
10        return true;
11    } finally {
12        lock.unlock();
13    }
14 }
```

4.1.1.12 本地方法栈溢出的情况

创建了过多的线程，线程独立，请求虚拟机栈和本地方法栈 – Stack Overflow

4.1.1.13 堆内存溢出

oom out of memory

申请的动态数据占据了过多的内存

4.1.1.14 HTTP请求头中有什么参数

请求报文

1. 请求方法 GET? POST
2. HTTP版本 1.1 2 3??
3. accept 期望接收的语言 zh en ??
4. user-agent: 访问者是通过什么工具来请求的
5. cache-control 是否强制刷新

相应报文

1. 状态码 200
2. 响应体一般是json 数据
3. content-type: 响应体里面的数据类型 image之类的

4.1.1.15 HTTP请求资源的方式

GET: 获取资源

POST: 创建资源

PUT: 创建(更新资源)

DELETE: 删除资源

4.1.1.16 如何防止SQL注入

使用预编译的方法来, 比如PreparedStatement类下面的setString方法来对参数进行处理, 简单来说

4.1.1.17 如何实现10000个qq判断是否在线的情况

java 中有BitSet这个类, 然后使用这个类来实现判断是否在线的情况

4.1.1.18 hashmap死锁产生情况

1.7 版本的死锁是在 rehash方法中的transfer方法产生的, 因为在扩容的过程中, 主要关于两个指针, e指针指向当前节点, next是e的下一个指针, 因为采用头插法会前后顺序调换, 导致产生换的现象.

4.1.1.19 谈谈对ConcurrentHashMap的扩容机制

1.7版本:

1. 1.7版本的ConcurrentHashMap是基于Segment分段实现的
 - *. Segment 依赖 ReentrantLock实现
 - *. 通过 hash 值和段数组长度-1 进行位运算确认当前 key 属于哪个Segment, 即确认其在 segments 数组的位置。
 - *. 再次通过 hash 值和 table 数组 (即 ConcurrentHashMap 底层存储数据的数组) 长度 - 1进行位运算确认其所在桶。
2. 每个Segment(数组)相对于一个小型的HashMap
3. 每个Segment内部会进行扩容, 和HashMap的扩容逻辑类似
4. 先生成新的数组, 然后转移元素到新数组中
5. 扩容的判断也是每个Segment内部单独判断的, 判断是否超过阈值

1.8版本

1. ConcurrentHashMap 不再基于Segment实现
2. 当某个线程运行put时候, 如果发现ConcurrentHashMap 正在进行扩容, 那么该线程一起进行扩容
3. 如果某个线程,put时, 发现并没有正在进行扩容, 则将keyvalue添加到ConcurrentHashMap中, 然后判断是否超过阈值, 超过则进行扩容
4. ConcurrentHashMap是支持多个线程同时扩容的
5. 扩容之前也先生成一个新的数组
6. 在转移元素时, 先将原数组分组, 将每组分给不同的线程来进行元素转移, 每个线程负责一组或多组的元素转移工作.

从JDK1.7版本的ReentrantLock+Segment+采用链表存储, 到JDK1.8版本中synchronized+CAS+Hash黑树/链表

concurrentHashMap 当初始化的时候出现了并发情况, 晚来的会使用线程礼让, 让第一个初始化的初始化完毕. 根据sizeCtl参数.

使用cas来保证对null节点放置元素.

使用8版本中synchronized对同一个数组的元素操作的时候.

我们看到 ($fh = f.hash$) == MOVED 有这样一个判断, MOVED 是一个成员静态变量, 值为-1, 当数组在扩容的时候会把数组的头节点的hash值变为-1, 所以当线程进来不管是查询还是修改还是添加只要看到当前主节点的hash值为-1时就会进入这里面的方法, 我们看到它里面是

<https://blog.csdn.net/wwj17647590781/article/details/118151008?spm=1001.2014.3001.5501>

4.1.1.20 hashTable

HashTable，它是线程安全的，它在所有涉及到多线程操作的都加上了synchronized关键字来锁住整个table，这就意味着所有的线程都在竞争一把锁，在多线程的环境下，它是安全的，但是无疑是效率低下的。

4.1.1.21 造成死锁的原因

1. 若干线程形成头尾相接的循环等待资源关系

解决方案:

1. 注意加锁的顺序, 保证每个线程按照同样的顺序进行加锁
2. 要注意加锁的时间, 可以针对锁设置一个超时时间
3. 要注意死锁检查, 这是一种预防机制, 确保在第一时间发现死锁并进行
4. 使用jstack 来查看dump 文件 <https://blog.csdn.net/u010647035/article/details/79769177>, 来查看锁的依赖关系
5. 避免一个线程使用多个锁
6. 尝试使用定时锁, 使用lock.tryLock(timeout)来代替使用内部锁
7. 对于数据库锁, 加锁和解锁必须在用一个数据连接里, 否则会出现锁失效的情况

4.1.1.22 深拷贝和浅拷贝

1. 一个对象中存在两种数据类型的属性, 一种是基本数据类型, 一种是实例对象的引用

A. 浅拷贝是指, 只会拷贝基本数据类型的值, 以及实例对象的引用地址, 并不会复制一份引用地址所指向的对象, 也就是浅拷贝出来的对象, 内部的属性指向的是同一个对象

B. 深拷贝是指, 既会拷贝基本数据类型的值, 也会针对实例对象的引用地址所指向的对象进行赋值, 深拷贝出来的对象, 内部的类执行指向的不是同一个对象

4.1.1.23 如果你提交任务时, 线程池队列已满, 这时会发生什么

1. 如果使用的是无界队列, 那么可以继续提交任务

2. 如果使用有界队列, 提交任务时, 如果队列满了, 如果线程数小于最大线程数, 那么增加线程, 如果线程数已经达到了最大值, 则使用拒绝策略进行拒绝

4.1.1.24 遇到过哪些设计模式

1. 代理模式, Spring中的AOP使用了代理模式
2. 工厂模式, Spring的BeanFactory就是一种工厂模式的实现

4.1.1.25 Spring中Bean是线程安全的吗?

Spring本身并没有针对bean做线程安全处理, 所以

1. 如果Bean是无状态的, 那么Bean则是线程安全的
2. 如果Bean是有状态的, 那么Bean则不是线程安全的

4.1.1.26 说说你了解的分布式锁实现

分布式锁所要解决的问题的本质是: 能够对分布在多台重启的线程对共享资源的互斥访问. 在这个原理上可以有很多的实现方式

1. 基于Redis, Redis中的数据也是在内存, 基于原子操作比如setnx
2. jmeter 压测工具
3. 使用setnx+过期时间, 使用 try + finally(来释放锁) 同时生成UUID(value), 自己使用的锁,自己释放, 防止自己的锁, 被别的进程释放
4. 补充, 自动延时()
5. redisson()

4.1.1.27 如何查看线程死锁

0. 使用ps H -eo pid,tid,%cpu|grep 2783 可以查看java进程中的线程
1. 使用jstack命令来查看
2. 对于mysql 使用select * from INFORMATION_SCHEMA.INNODB_LOCKS 查看正在锁的事物
3. 查看等待锁的事物 SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS

4.1.1.28 线程之间如何进行通讯的

1. 使用共享内存或基于网络来进行通信
2. 如果是通过共享内存来进行通信

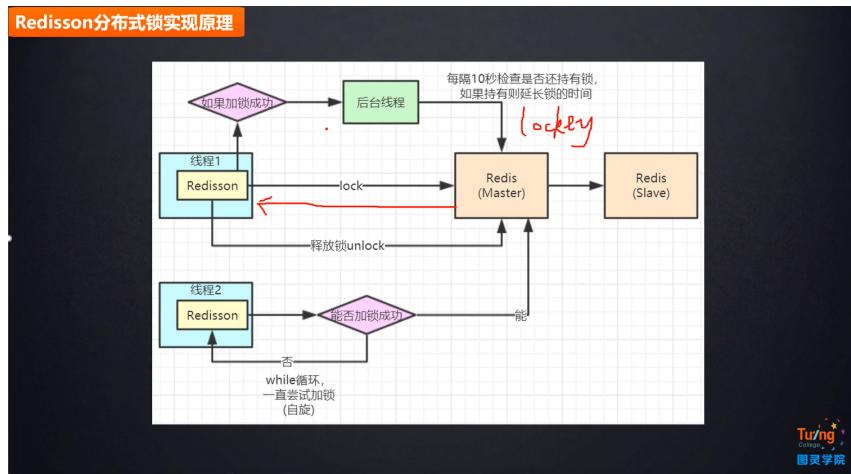


图 4-2 redission

4.1.1.29 快速失败(fail-fast) 和安全失败(fail-safe) 的区别是什么?

1. java.util包下面的所有的集合类都是快速失败的, 而java.util.concurrent包下面的所有类都是安全失败的. 快速失败的迭代器会抛出ConcurrentModificationException异常, 而安全失败的迭代器永远不会抛出这样的异常.

4.1.1.30 异常

- (1) Throwable(可抛出) 超类, 有两个子类Error 和 exception 错误和异常

4.1.1.31 synchronized的底层实现细节

1. synchronized作用

原子性: synchronized保证语句内操作是原子的

可见性: synchronized保证可见性(通过在执行unlock之前, 必须先报此变量同步回主内存实现)

有序性: synchronized保证有序性(通过"一个变量在同一时刻只允许一条线程对其进行lock操作")

可见性补充: 其实真正解决这个问题的是JMM关于Synchronized的两条规定:

- 1、线程解锁前, 必须把共享变量的最新值刷新到主内存中;
- 2、线程加锁时, 讲清空工作内存中共享变量的值, 从而使用共享变量是需要从主内存中重新读取最新的值 (加锁与解锁需要统一把锁)

4.1.1.32 线程池参数

ThreadPoolExecutor的创建参数:

- (1) corePoolSize, 核心运行的线程个数, 若线程池已创建的线程数小于corePoolSize, 即使此时存在空闲线程, 也会通过创建一个新线程来执行该任务.
- (2) maximumPoolSize: 最大线程个数, 当大于这个值就会将准备新加入的异步任务有一个丢弃处理机制来处理, 大于corePoolSize且小于maximumPoolSize存入等待队列,
- (3) workQueue: 任务等待队列, 当达到corePoolSize的时候就向该等待队列放入线程信息.
- (4) keepAliveTime: 默认0, 当线程没有任务处理后空闲线程保持多长时间, 不推荐使用, 一般会中止超过corePoolSize数量的线程资源, 空闲线程时间超过keepAliveTime, 线程将会被回收
- (5) threadFactory: 构造Thread方法, 使用默认的default实现.
- (6) defaultHandler: 当maximumPoolSize达到后丢弃处理的方法实现, java默认是抛出异常.

4.1.1.33 你觉得核心线程数和最大线程数之间应该如何设置呢

1. 对于CPU密集型任务, 由于CPU密集型任务的性质, 导致CPU的使用率很高, 如果线程池中的核心线程数量过多, 会增加上下文切换的次数, 带来额外的开销。因此, 一般情况下线程池的核心线程数量等于CPU核心数+1。
2. 对于I/O密集型任务, 由于I/O密集型任务CPU使用率并不很高, 可以让CPU在等待I/O操作的时去处理别的任务, 充分利用CPU。因此, 一般情况下线程的核心线程数等于2*CPU核心数。

4.1.1.34 synchronized和ReentrantLock的区别

1. synchronized是一个关键字, ReentrantLock是一个类
2. synchronized会自动的加锁和释放锁, ReentrantLock是一个类
3. synchronized的底层是jvm层面的锁, ReentrantLock是API层面的锁
4. synchronized是非公平锁, ReentrantLock可以选择公平锁或非公平锁
5. synchronized锁的是对象, 锁信息保存在对象头中, ReentrantLock锁的是线程
6. synchronized底层有一个锁升级的过程

4.1.1.35 线程池ThreadPoolExecutor中使用的BlockQueue

(1) 直接提交队列: 简单来说使用SynchronousQueue队列, 提交的任务不会被保存, 总是会马上提交执行。如果用于执行任务的线程数量小于maximumPoolSize, 则尝试创建新的进程, 如果达到maximumPoolSize设置的最大值, 则根据你设置的handler执行拒绝策略。因此这种方式你提交的任务不会被缓存起来, 而是会被马上执行, 在这种情况下, 你需要对你程序的并发量有个准确的评估, 才能设置合适的maximumPoolSize数量, 否则很容易就会执行拒绝策略;

(2) 有界的任务队列可以使用ArrayBlockingQueue实现, 若有新的任务需要执行时, 线程池会创建新的线程, 直到创建的线程数量达到corePoolSize时, 则会将新的任务加入到等待队列中。若等待队列已满, 即超过ArrayBlockingQueue初始化的容量, 则继续创建线程, 直到线程数量达到maximumPoolSize设置的最大线程数量, 若大于maximumPoolSize, 则执行拒绝策略。在这种情况下, 线程数量的上限与有界任务队列的状态有直接关系, 如果有界队列初始容量较大或者没有达到超负荷的状态, 线程数将一直维持在corePoolSize以下, 反之当任务队列已满时, 则会以maximumPoolSize为最大线程数上限。

(3) 使用无界任务队列, LinkedBlockingQueue 实现线程池的任务队列可以无限制的添加新的任务, 而线程池创建的最大线程数量就是你corePoolSize设置的数量, 也就是说在这种情况下maximumPoolSize这个参数是无效的, 哪怕你的任务队列中缓存了很多未执行的任务, 当线程池的线程数达到corePoolSize后, 就不会再增加了; 若后续有新的任务加入, 则直接进入队列等待, 当使用这种任务队列模式时, 一定要注意你任务提交与处理之间的协调与控制, 不然会出现队列中的任务由于无法及时处理导致一直增长, 直到最后资源耗尽的问题。

(4) 优先任务队列: 优先任务队列通过PriorityBlockingQueue实现,

4.1.1.36 Executors 工厂类实现线程池

通过创建不同的ThreadPoolExecutor参数.

(1) FixedThreadPool 定长, corePoolSize == maximumPoolSize

(2) SingleThreadExecutor 单一线程无界队列

以上两种可能会堆积大量的请求, 从而引起OOM异常

(3) CachedThreadPool 采用maximumPoolSize为无限大, 容易创建大量线程, 从而耗尽系统资源.

submit 基于 Future 来包装返回值对象, 使用callable来进行调用有返回值

execute 基于 runnable 来实现线程池, 没有返回值

```

1  final ExecutorService service = Executors.newFixedThreadPool(5);
2  Future<String> f = service.submit(new Callable<String>() {
3      @Override
4      public String call() throws Exception {
5          Thread.sleep(3000);
6          System.out.println("方法执行了call");
7          return "方法返回值call";
8      }
9  });
10 System.out.println(f.get());
11 service.execute(new Runnable() {
12     @Override
13     public void run() {
14         try {
15             Thread.sleep(1500);
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19         System.out.println("fdasfdsa");
20         return ;
21     }
22 });
23 service.shutdown();

```

4.1.1.37 线程池的拒绝策略

- (1) `abortPolicy` 默认: 直接抛出异常
- (2) `CallerRunsPolicy`: 直接调用主线程来执行任务.
- (3) `DiscardPolicy`: 不能执行的任务被删除, 和`abortPolicy`一样, 但是不抛出异常.
- (4) `DiscardOldestPolicy`: 位于工作队列头部的任务将被删除, 然后重新执行程序.

4.1.1.38 8种基本数据类型

4.1.1.39 Comparable & Comparator 区别

`Comparable` 是接口能力赋予

```

1  public interface Comparable<T> {
2      public int compareTo(T o);

```

表 4-1 实现

类型	大小(注释/包装类)
byte	8(Byte)
short	16(Short)
int	32(Integer)
long	64(Long)
float	32(Float)
double	64(Double)
char	16(Character)
boolean	8(Boolean)

3 }

Comparator 是外部比较器, 也是接口, 类似于 C++sort 中自定义的cmp函数

```

1 Collections.sort(list, new Comparator<Person2>() {
2     public int compare(Person o1, Person o2) {
3         return o1.getAge() - o2.getAge();
4     }
5 })

```

4.1.1.40 java采用值传递还是引用传递?

采用值传递, 但是因为采用浅拷贝, 所以会修改传递的对象的相关属性.

4.1.1.41 java深拷贝和浅拷贝

实现了Cloneable接口实现深拷贝.

4.1.1.42 java"==" 和 equals 的区别

1. "==" : 如果是基本数据类型, 则直接对值进行比较, 如果是引用数据类型, 则是对他们的地址进行比较;

2. equals方法继承Object类, 在具体实现时可以覆盖父类中的实现. 看一下Object中equals的源码发现, 它的实现也是对对象的地址进行比较, 可以覆盖实现这个方法, 如果两个对象的类型一致, 并且内容一致, 则返回true.

在实际开始中总结:

(1) 类未复写equals, 则使用equals方法比较两个对象时, 相当于==比较, 及两个地址是否相等. 地址相等, 返回true, 地址不相等, 返回false.

(2) 类复写equals方法, 走复写之后的判断方式. 通常, 我们会将equals复写成: 当两个对象内容相同时, 则equals返回true, 内容不同时, 返回false.

对于set, hashMap, hashset等, 还要重写hashCode值, 比如set判断两个元素是否相等的时候, 会判断hashcode和equals都相等, 则认为相等, 不会添加新元素.

4.1.1.43 String和StringBuilder, StringBuffer的区别

String是不可变字符串对象(final的char数组), StringBuilder和StringBuffer(线程安全)是可变字符串对象.

为什么String是final修饰的?

1. 为了实现字符串池, 因为只有当字符串是不可变的, 字符串池才有可能实现.

4.1.1.44 Java反射机制

简单来说就是在, 运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法; 对于任意一个对象, 都能够调用它的任意方法和属性; 并且能改变它的属性. 这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制.

优点: 代码灵活度提高

缺点: 性能瓶颈, 性能较慢.

4.1.1.45 简述面向对象三大特征, 继承, 封装, 多态

1. 封装

简单来说, 就是使用private方法将没有必要暴露的方法和属性进行隐藏.

2. 继承

继承是从已有的类中派生出行的类, 减少代码冗余.

3. 多态

父类引用指向不同子类对象.

4.1.1.46 多态

一般使用instance of 来判断对象的子类关系. 增加向下转型的健壮度.

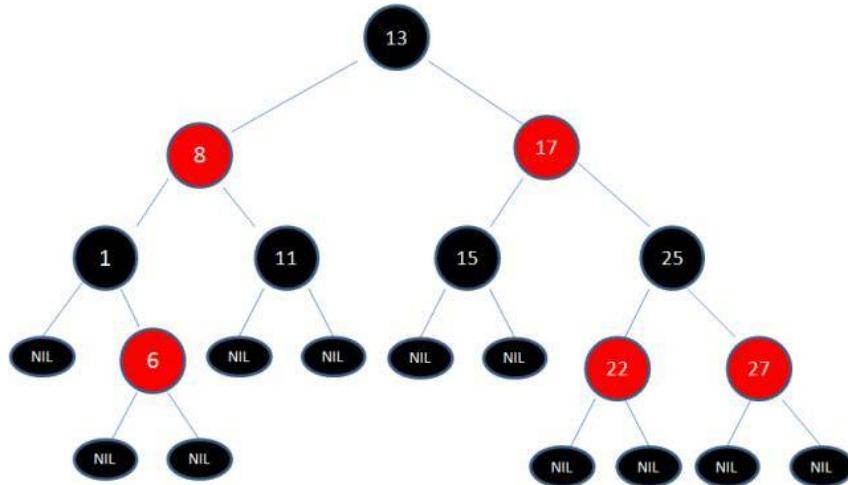


图 4-3 redblack

4.1.1.47 内部类

- (1) 静态内部类访问外部变量必须是静态的.
- (2)

4.1.1.48 红黑树

一般考察红黑树: 只考察概念.

1. 节点是红色或黑色
2. 根节点是黑色
3. 所有叶子都是黑色(叶子是NIL节点).
4. 每个红色节点必须有两个黑色节点
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点.

4.1.1.49 hashmap的数据结构

1. jdk1.7 由数组 + 链表来构成
2. jdk1.8 由数组 + 链表 + 红黑树来构成
3. jdk1.8的时候, 当元素不超过64个的时候, 不会出现链表转红黑树, 当元素超过64个的时候, 会出现链表转红黑树.
4. jdk1.8 当链表长度达到8个的时候, 链表会转为红黑树, 当红黑树元素长度退回到6个的时候会出现红黑树转为链表.
5. jdk1.7 采用头插法, jdk1.8采用尾插法.

4.1.1.50 hashmap的put方法

1. 根据Key通过哈希算法与与运算得到数组下标
 2. 如果数组下标位置元素为空, 则将key和value封装为Entry对象并放入该位置
 3. 如果数组下标元素不为空
 - 1.7, 则先判断是否需要扩容, 如果要扩容就进行扩容, 如果不用扩容就生成Entry对象, 并使用头茶法添加到当前位置的链表中
 - 1.8 先判断当前位置上Node的类型, 看是红黑树Node还是链表Node
 - a. 如果是红黑树Node, 则将key和value 封装为一个红黑树节点并添加到红黑树中
 - b. 如果是链表节点, 使用尾插法插入到链表的最后位置去, 插入完后会判断当街链表的个数看是否需要转为红黑树(超过8个)元素.
 - c. 判断是否需要扩容(0.75*16默认值), 需要扩容就扩容, 不需要就结束PUT方法
- JDK8则因为巧妙的设计, 性能有了大大的提升: 由于数组的容量是以2的幂次方扩容的, 那么一个Entity在扩容时, 新的位置要么在原位置, 要么在原长度+原位置的位置。

4.1.1.51 介绍一下ThreadLocal

1. ThreadLocal 是java中所提供的的线程本地存储机制, 可以利用该机制将数据缓存在某个线程内部, 该线程可以在任意时刻, 任意方法中获取缓存的数据
2. ThreadLocal 底层是通过ThreadLocalMap来实现的, 每个Thread对象中都存在一个ThreadLocalMap, Map的key为ThreadLoacl对象, Map的value为需要缓存的值
3. 如果在线程池中使用ThreadLocal会造成内存泄露, 因为当ThreadLocal对象使用完后, 应该要报设置的key, value 也就是Entry对象进行回收, 但线程池中的线程不会回收, 而线程对象是通过强引用指向ThreadLocalMap, ThreadLocalMap也是通过强引用指向Entry对象, 线程不被回收, Entry对象也就不会被回收, 从而出现内存泄露, 解决方法是, 在使用了ThreadLocal对象之后, 手动调用ThreadLocal的remove方法, 手动清除Entry对象.

4.1.1.52 heap和stack有什么区别

1. java的内存分为两类, 一类是堆内存, 一类是栈内存
2. 栈内存是指程序进入一个方法时, 会为这个方法单独分配一块私属存储空间, 用于存储这个方法内部的局部变量. 当这个方法结束时, 分配给这个方法的栈会释放, 这个栈中的变量也随之释放.

3. 使用new创建的对象存放在堆里, 不会随方法的结束而消失. 方法中的局部变量使用final修饰后, 放在堆中, 而不是栈中.

4.1.1.53 Array 和 ArrayList 的区别

1. Array 大小固定(int a[] = {1, 2, 3}), ArrayList 大小是动态变化的.

transient: 将不需要序列化的属性前添加关键字transient, 序列化对象的时候, 这个属性就不会被序列化。

默认初始化大小是10个容量.

```
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

新的大小是旧的大小的1.5倍. 扩容会调用Arrays.copyOf操作, 然后申请新的大小.

4.1.1.54 Java各种锁: 悲观锁, 泪管所, 自旋锁, 偏向锁, 轻量/重量锁, 读写锁, 可重入锁

悲观锁和乐观锁指的是并发情况下的两种不同策略, 是一种宏观的描述.

1. 悲观锁和乐观锁指的是并发情况下的两种不同策略, 是一种宏观的描述.

4.1.1.55 Collection 和 Collections 的区别

1. Collection 是集合类的上级接口, 继承他的接口主要是set和list
2. Collections 类数针对集合类的一个帮助类. 它提供了一系列的静态方法对各种集合的搜索, 排序, 线程安全化等操作.

4.1.1.56 接口与抽象类区别

1. 类可以实现多个接口但只能继承一个抽象类
2. 接口中变量被隐性制定为public static final, 方法被指定为 public abstract
3. 接口里面所有的方法都是Public的, 抽象类允许Private, Protected方法
4. JDK接口可以实现默认方法和静态方法, 前面加default, static关键字.
5. 设计层面: 抽象类是对事物的抽象, 接口是对行为的抽象.

```
1 public interface InterfaceJDK8 {
2
3     // 接口的普通抽象方法 /**
4     public void common(String str);
5 }
```

```
6  /*jdk1.8 默认方法：  
7   允许在已有的接口中添加新方法，而同时又保持了与旧版本代码的兼容性，  
8   默认方法与抽象方法不同之处在于抽象方法必须要求实现，但是默认方法则没有要求实现，  
9   相反，接口提供了一个默认实现，这样所有的接口实现者将会默认继承他  
10  （如果有必要的话，可以覆盖这个默认实现）。  
11  接口的默认方法：得到接口的实现类对象，直接用对象的引用方法名。默认方法可以被实现  
12  类覆盖。.  
13  */  
14  default public void defaultMethod(String str){  
15      System.out.println("InterfaceJDK8:" + str);  
16  }  
17  /*jdk1.8 静态方法：  
18  允许在已有的接口中添加静态方法，接口的静态方法属于接口本身，不被继承，也需要提供  
19  方法的现。  
20  */  
21  public static void staticMethod(String str){  
22      System.out.println("InterfaceJDK8:" + str);  
23  }  
24 }
```

4.1.1.57 ArrayList和LinkedList内部实现大致是怎样的? 他们之间的区别和优缺点

1. **ArrayList:** 内部使用数组的形式实现了存储, 利用数组的小表进行元素的访问, 因此对元素的随机访问速度非常快. 初始化大小为10, 插入新元素的时候, 会判断是否需要扩容, 扩容的步长是0.5倍原容量, 扩容方式是利用数组的复制, 因此有一定的开销
2. **LinkedList:** 内部使用双向链表的结构实现存储, **LinkedList**有一个内部类作为存放元素的单元, 里面有三个属性, 用来存放元素本身以及前后2个单元的引用, 另外**LinkedList**内部还有一个Header属性, 用来标识起始位置, **LinkedList**的第一个单元和最后一个单元都会指向header, 因此形成了一个双向链表结构.
3. **LinkedList**还额外实现了**Deque**接口, 所以**LinkedList**还可以当做队列来使用.

4.1.1.58 ==和equals的区别

==是运算符, 而**equals**是**Object**的基本方法, **==**用于基本数据的类型比较, 或者是比较两个对象的引用是否相同, **equals**用于比较两个对象的值是否相等, 例如字符串的比较.

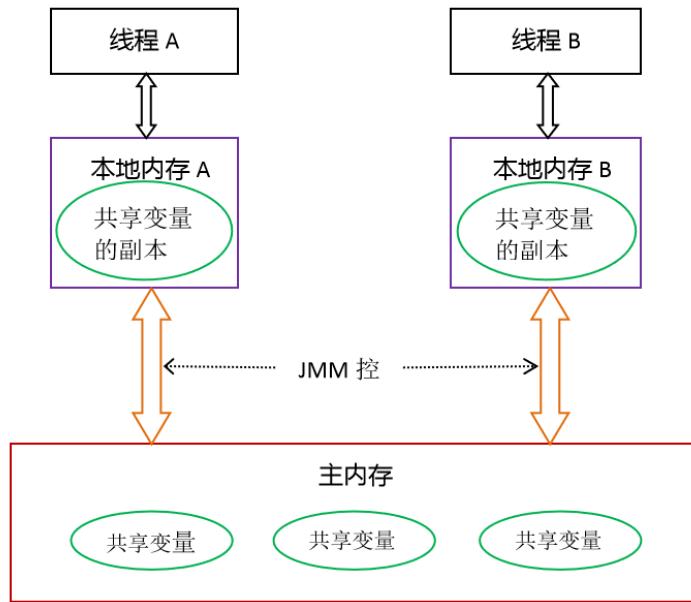


图 4-4 JMM

4.1.1.59 hashCode方法的作用

1. 如果两个对象equals方法相等, 那么hashCode一定相同
2. 如果两个对象的hashCode相同, 并不表示两个对象相同(只能表示hash碰撞相同), equals方法相同.

4.1.1.60 反射

简单来说, 在运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法, 对于任意一个对象, 都能够调用他的任意方法和属性, 并且能够改变他的属性.

4.1.1.61 简述Java内存模型(JMM)

简单来说就是, java中存在一个主内存, java中所有变量都存在主内存中, 对所有线程进行共享, 而每个线程又存在自己工作内存, 工作内存存储的是主存中某些变量的拷贝, 线程对所有变量的操作并非发生在主存区, 而是发生在工作内存中, 线程之间是不能直接相互访问, 变量在程序中的传递主要依赖主存完成.

4.1.1.62 Java内存模型中的可见性, 原子性和有序性

可见性, volatile

原子性, 各种锁

有序性, 线程内有序

4.1.1.63 happen-before原则

虽然有好几个, 但基本上描述模糊的就不写了

- (1) 锁的happen-before, 就是同一个锁的unlock操作happen-before此锁的lock操作.
- (2) 传递性: A hb b, b hb c; A happen-before C;
- (3) 对象的构造函数在finalize方法之前.

4.1.1.64 wait/notify, await/singal

Condition 的 await, signal, singalAll 与 Object 的 wait, notify, notifyAll 都可以实现的需求, 两者在使用上也是非常类似, 都需要先获取某个锁之后才能调用, 而不同的是 Object wait,notify 对应的是 synchronized 方式的锁, Condition await, singal 则对应的是 ReentrantLock (实现 Lock 接口的锁对象) 对应的锁

下方是Condition的示例

4.1.1.65 多线程wait和sleep区别

- (1) 主要在获得执行权和释放锁之间的区别, wait会释放执行权, 然后释放锁, sleep 只会释放执行权
- (2) 如果notifyAll() 如果有多个线程在等待, 只会有一个线程获得执行权.

4.1.1.66 Collection<? extends Person> s

这叫泛型上限, 这样取出都是按照上限类型来运算的. 不会出现安全隐患

```

1 public class Message {
2     /** 当前消息数量*/
3     private int count = 0;
4     /** 信息存放最大限数*/
5     private int maximum = 20;
6     /** 重入锁*/
7     private Lock lock;
8     /** 生产者锁控制器*/

```

```
9  private Condition producerCondition;
10 /* 消费者锁控制器*/
11 private Condition consumerCondition;
12
13 public Message() {}
14
15 public Message(final Lock lock, final Condition producerCondition,
16   final Condition consumerCondition) {
17   this.lock = lock;
18   this.producerCondition = producerCondition;
19   this.consumerCondition = consumerCondition;
20 }
21 /**
22 * 生产消息
23 */
24 public void set() {
25   /** 获得锁*/
26   lock.lock();
27   try {
28     if (count <= maximum) {
29       /** 生产一个消息*/
30       System.out.println("生产者(" + Thread.currentThread().getName() + "线程" + Thread.currentThread().get
31       "程" + Thread.currentThread().getName() + "生产了一个消息", 当前有" + (++count) + "个消息");
32       /** 唤醒等待的消费者*/
33       consumerCondition.signal();
34     } else {
35       try {
36         /**
37          * 如果当前消息大于 信息最大数maximum
38          * 生产者进入睡眠等待状态/
39          */
40         producerCondition.await();
41         System.out.println("生产者(" + Thread.currentThread().getName() + "线程" + Thread.currentThread().get
42         "程" + Thread.currentThread().getName() + "进入睡眠");
43       } catch (InterruptedException e) {
44         e.printStackTrace();
45       }
46     }
47     /**
48      */
49   }
}
```

```
50
51     /**
52      * 消费消息
53      */
54     public void get() {
55         /** 获取锁*/
56         lock.lock();
57         try {
58             if (count > 0) {
59                 /** 消费一个消息*/
60                 System.out.println消费者(" 线
程" + Thread.currentThread().getName() + "消费了一个消息", 当前
有" + (--count) + "个消息");
61                 /** 唤醒等待的生产者*/
62                 producerCondition.signal();
63             } else {
64                 try {
65                     /** 如果没有消息, 消费者进入睡眠等待状态*/
66                     consumerCondition.await();
67                     System.out.println消费者(" 线
程" + Thread.currentThread().getName() + "进入睡眠");
68                 } catch (InterruptedException e) {
69                     e.printStackTrace();
70                 }
71             }
72         } finally {
73             /** 释放锁*/
74             lock.unlock();
75         }
76     }
77
78 }
79
80 public class Producer implements Runnable {
81     private Message message;
82     public Producer(Message message) {
83         this.message = message;
84     }
85
86     @Override
87     public void run() {
88         while(true) {
89             try {
90                 Thread.sleep(500);
91             } catch (InterruptedException e) {
```

```
92         e.printStackTrace();
93     }
94     message.set();
95   }
96 }
97
98 }
99 public class Consumer implements Runnable {
100     private Message message;
101     public Consumer(Message message) {
102         this.message = message;
103     }
104
105     @Override
106     public void run() {
107         while(true) {
108             try {
109                 Thread.sleep(1000);
110             } catch (InterruptedException e) {
111                 e.printStackTrace();
112             }
113             message.get();
114         }
115     }
116 }
117 }
118 import java.util.concurrent.locks.Condition;
119 import java.util.concurrent.locks.Lock;
120 import java.util.concurrent.locks.ReentrantLock;
121
122 public class App {
123     public static void main(String[] args) {
124         /** 重入锁*/
125         final Lock lock = new ReentrantLock();
126         /** 生产者锁控制器*/
127         final Condition producerCondition = lock.newCondition();
128         /** 消费者锁控制器*/
129         final Condition consumerCondition = lock.newCondition();
130         final Message message = new Message(lock, producerCondition,
131         consumerCondition);
132         /** 建几个生产线程*/
133         new Thread(new Producer(message)).start();
134         new Thread(new Producer(message)).start();
135         new Thread(new Producer(message)).start();
```

```

135     /** 建几个消费线程*/
136     new Thread(new Consumer(message)).start();
137     new Thread(new Consumer(message)).start();
138     new Thread(new Consumer(message)).start();
139     new Thread(new Consumer(message)).start();
140 }
141
142 }

```

4.1.1.67 线程的状态有哪些?

- (1) 新建状态(NEW): 线程创建之后
- (2) 可运行(RUNNING): 可能正在运行, 也可能正在等待时间片
- (3) 阻塞(BLOCKED): 等待获取一个排它锁, 如果期限陈释放了锁就会结束此状态.
- (4) 无线等待(WAITING): 等待其他线程显式地唤醒, 否则不会被分配CPU时间片片
- (5) 限期等待(TIME_WAITING): 如果没人唤醒在一定时间内系统会自动唤醒
- (6) 终止(TERMINATED): 可以是线程结束任务之后自己结束, 或者产生了异常而结束

线程创建之后处于New状态, 调用start()方法后开始运行, 线程这时候处于Ready可运行状态. 可运行状态的线程获得cpu时间片后就处于RUNNING状态. 当线程执行wait()方法之后, 线程进入WAITING(等待)状态. 进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态, 而TIME_WAITING超时等待状态相当于在等待状态的基础上增加了超时限制, 比如通过SLEEP方法或wait放假将java线程至于TIME_WAITING状态, 到超时之后, java线程将会发挥RUNNABLE状态. 当线程调用同步方法是在没有获取到所的情况下, 线程将会进入到BLOCK状态. 执行完Runnable的run()方法之后将会进入到TERMINATED状态.

4.1.1.68 进程的状态有哪些?

就绪状态: 已获除CPU以外所需的资源, 等待分配处理机资源

运行状态: 占用处理机资源运行

阻塞状态: 进程等待某种条件, 在满足之前无法执行

new 行启动进程获除CPU以外的资源被准许(admitted)进入就绪状态ready, 就绪进程获得系统分配CPU资源后背调度器调度(scheduler dispatch)进入运行态running, 运行态进程在执行完任务后退出(exit)进程终止terminated.

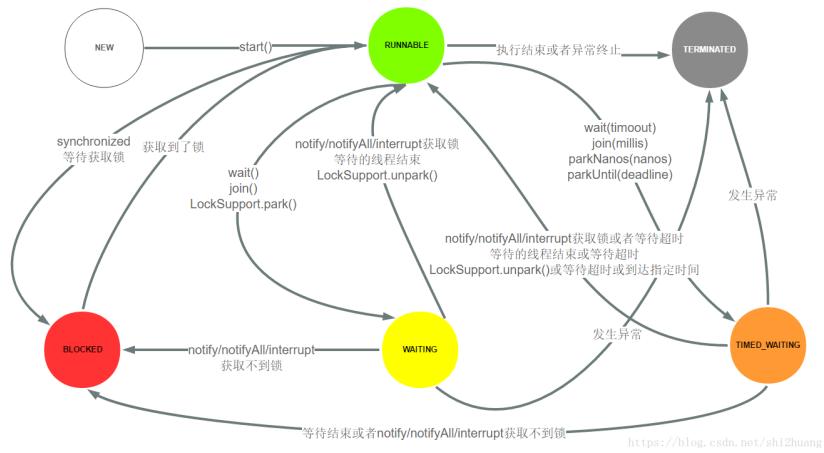


图 4-5 javaThreadState

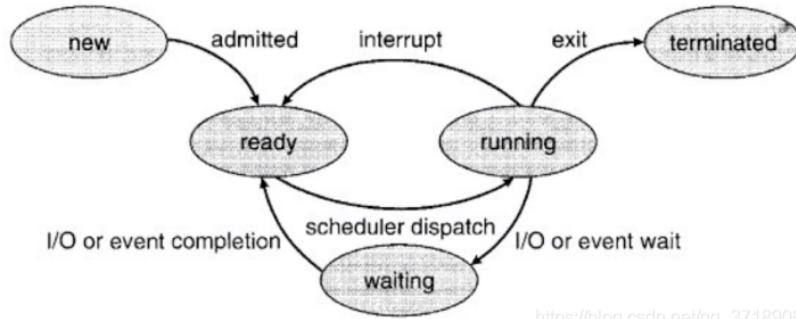


图 4-6 processorstatus

运行太进程当时间片用完时候会先中断(interrupt)进入到就绪状态, 等待下次时间片轮转分配CPU资源. 运行太进程当遇到等待用户输入或时间等待(I/O or event wait)会进入阻塞态waiting, 阻塞态进程IO输入完毕或时间完成(I/O or event completion)会进入到就绪态等待系统分配CPU资源.

4.1.1.69 创建线程的几种方式

(1) 继承Thread类创建线程

定义Thread类的子类, 并重写该类的run方法

创建实例

调用实例start()方法

(2) 实现Runnable接口创建线程

实现一个接口类, 重新run方法.

创建Runnable实现类实例, 并以此实例作为Thread的target来创建Thread对象, 该Thread对象才是真正的线程对象.

调用线程对象start方法来启动该线程

(3) 使用Callable和Future创建线程: 与Runnable相比Callable是有返回值的, 返回值通过FutureTask进行封装

创建Callable接口的实现类, 并实现call()方法, 该call()方法将作为线程执行体, 兵器有返回值

创建Callbale实现类实例, 使用FutureTask类来包装Callable对象, 该FutureTask对象封装了该Callable对象的call()方法的返回值.

使用FutureTask对象作为Thread对象的target创建biang启动新线程

调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

(4) 使用线程池例如Executor框架(工厂方法)

(5) 创建线程的方式的对比

1. Runnable 不可以抛出异常, Callable可以

2. Runnable 不可以有返回值, Callable 通过封装FutureTask 可以拿到返回值

4.1.1.70 synchronized锁升级: 无锁, 偏向锁, 轻量级锁, 重量级锁(与锁的优化一起学习)

这个叫做锁的膨胀.

(1) 偏向锁, 初次执行到synchronized代码块的时候, 锁对象变成偏向锁, 通过CAS修改对象头里的锁标志位, 字面意思是"偏向于第一个获得它的线程"的锁. 会存储获取锁的线程的地址. 偏向锁解锁, 不需要修改对象头的markword, 减少了一次CAS操作, 锁不会释放, 但是遇到冲突, 会由JVM来进行判断升级. 执行完同步代码块之后, 线程并不会主动释放偏向锁, 当第二次达到同步代码块时, 线程会判断此时持有锁的线程是否就是自己(持有锁的线程ID也在对象头里), 如果是正常往下执行. 由于之前没有释放锁, 这里也就不需要重新加锁. 如果自始至终使用锁的线程只有一个, 很明显偏向锁几乎没有额外开销, 性能极高.

(2) 轻量级锁, 自旋锁, 地担忧第二个线程加入锁竞争, 偏向锁, 就升级为轻量级锁. 只有当某线程获取锁的时候, 发现该锁已经被占用, 只能等待其实方, 这才发生了锁的竞争. 在所竞争下, 没有抢到锁的线程将自旋, 即不停的循环判断锁是否能够被成功获取. 长时间的自选操作是非常消耗资源的, 一个线程持有锁, 其他线程就只能在原地空号CPU. 如果达到某个最大自旋次数, 会将轻量级锁升级为重量级锁. 当后续线程尝试获取锁时, 直接将自己挂起.

- (3) 偏向锁, 假定条件只有一个线程去获取锁
- (4) 轻量级锁, 假定条件是多个线程交替去获取锁

4.1.1.71 如何使用synchronized

- 1. 普通同步方法

5

JVM

► ...

5.1 知识点和方法论

5.1.1 知识点

5.1.1.1 新生代复制算法的使用

平均分成A/B块太浪费内存，采用Eden/S0/S1三个区更合理，一个较大的Eden空间和两个较小的Survivor空间，空间比例为Eden:S0:S1==8:1:1，有效内存（即可分配新生对象的内存）是总内存的9/10。

算法过程：

1. Eden+S0可分配新生对象；
2. 对Eden+S0进行垃圾收集，存活对象复制到S1。清理Eden+S0。一次新生代GC结束。
3. Eden+S1可分配新生对象；
4. 对Eden+S1进行垃圾收集，存活对象复制到S0。清理Eden+S1。二次新生代GC结束。
5. goto 1。

5.1.1.2 说一下JVM中，哪些是共享区，哪些可以作为gc root

共享区：方发区和堆

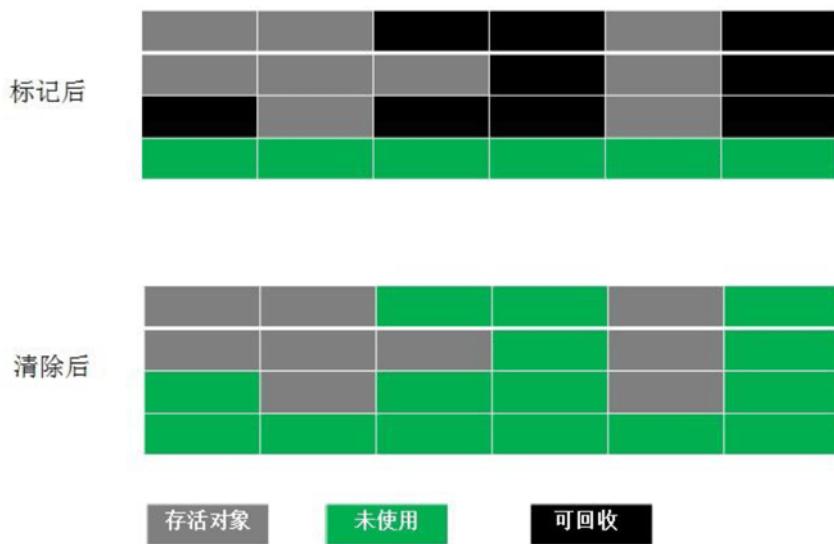


图 5-1 signremove

每个线程独有: 栈, 本地方法栈, 程序技术器

堆中, 从gc root可以找到一连串的对象, 就是正常对象, 没有被找到的对象可以被回收.

5.1.1.3 你们项目如何排查JVM问题

1. 使用jvisualvm图形化查看内存的变化, 发现频繁的fullgc, 但是并没有出现oom现象, 可能是年轻代的内存不够, 对于大对象如果新生代放不下会直接放入老年代, 导致频繁fullgc, 通过增大新生代内存, fullgc减少, 证明修改有效.
2. 对于已经发生了oom异常的, 生成dump文件(-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/local/base)

使用jvisualvm等工具来分析dump文件, 更具dump文件找到异常的实例对象, 和异常的线程, 定位到具体的代码, 然后再进行详细的分析和调试

5.1.1.4 GC的三种收集方法: 标记清除, 标记整理, 复制算法的原理与特点, 分别用在什么地方, 如果让你优化收集方法, 有什么思路

1. 标记清除: 先标记, 标记完毕之后再清除, 缺点: 效率不高会产生碎片.
2. 标记整理: 标记完毕之后, 让所有存活的对象向一端移动
3. 复制算法: 分别8:1的Eden去和survivor区
4. 分代收集算法-重点

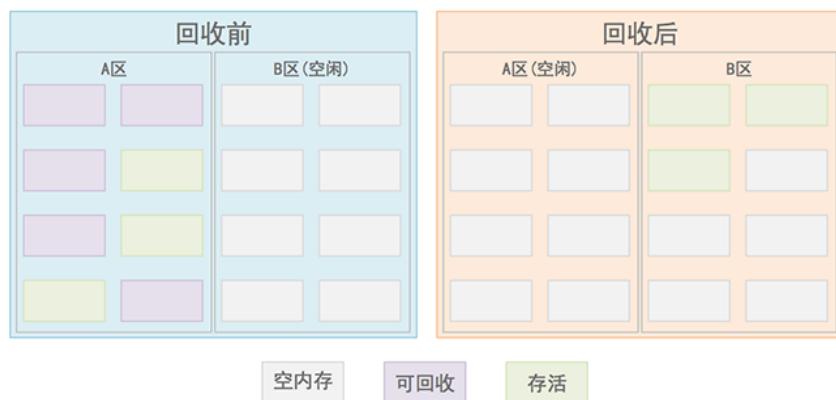


图 5–2 jvmcopy

一般将Java分成新生代和老年代, 新生代使用复制算法, 老年代使用标记整理算法.

5.1.1.5 JVM的主要组成部分及其作用?

JVM包含两个子系统和两个组件, 两个子系统为Class loader(类加载), Execution engine(执行引擎); 两个组件为 Runtime data area(运行时数据区), native Interface(本地接口)

1. Class loader: 根据给定的完全限定类名(如:java.lang.object)来装在class文件到Runtime data area中的method area.
2. Execution engine(执行引擎): 执行classes中的指令
3. native Interface(本地接口): 与native libraries交互, 是其他编程语言交互的接口.
4. Runtime data area(运行时数据区): 这就是我们常说的jvm的内存

作用: 首先通过编译器把java代码转换成字节码, 类加载器(ClassLoader)再把字节码加载到内存中, 将其放在运行时数据区(Runtime data area)的堆区内, 而字节码文件知道jvm的一套指令集规范, 并不能直接交给底层操作系统去执行, 因此需要特定的命令解析器执行引擎(Execution Engine), 将字节码翻译成底层系统指令, 在交由CPU去执行, 而这个过程中需要调用其他语言的本地库接口(Native Interface)来实现整个程序的功能.

Java程序运行机制步骤

1. 编码: IDEA等IDE进行编码java, 后缀.java
2. 编译: javac 将源代码编译成字节码文件, 字节码文件的后缀名为.class

类的加载是将类的.class文件中的二进制数据读入到内存中, 将其放在运行时数据区的方法区内, 然后在堆区创建一个java.lang.Class对象, 用来封装类在堆区内的数据结构.

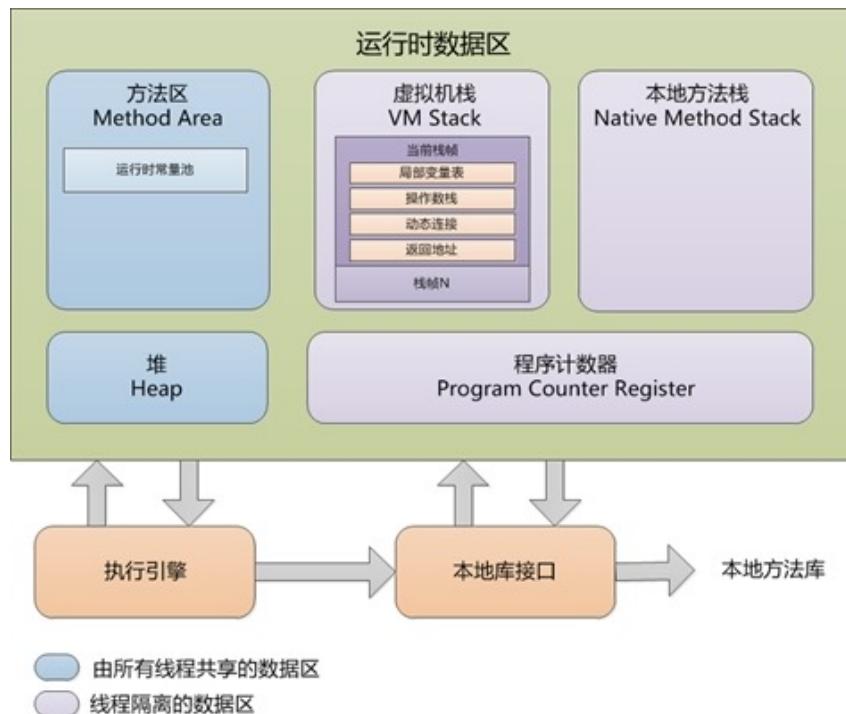


图 5-3 jvm

5.1.1.6 JVM运行时数据区

运行时数据区由如下几个区域构成

1. 程序计数器(PC): 当前线程所执行字节码的行号指示器, 字节码解析器的工作是通过改变这个计数器的值, 来选去下一条需要执行的字节码指令.
2. java虚拟机栈(Java Virtual Machine Stacks): 用于存储局部变量表, 操作数栈, 动态链接, 方法出口等信息.
3. 本地方法栈(Native Method Stack) : 与虚拟机栈的作用是一样的, 只不过虚拟机栈是服务Java方法的, 而本地方法栈是为虚拟机调用Native方法服务的.
4. Java堆(Java Heap): Java 虚拟机中内存最大的一块, 是被所有线程共享的, 几乎所有的对象实例, 都在这里分配内存;
5. 方法区(Method Area): 用于存储已被虚拟机加载的类信息, 常量, 静态变量, 及时编译后的代码等数据.

5.1.1.7 JVM运行时数据区这些方法的关系

可以看到PC指针和虚拟机栈和本地方法栈是线程独有的. 而堆,方法区和运行时常量池是属于线程共享

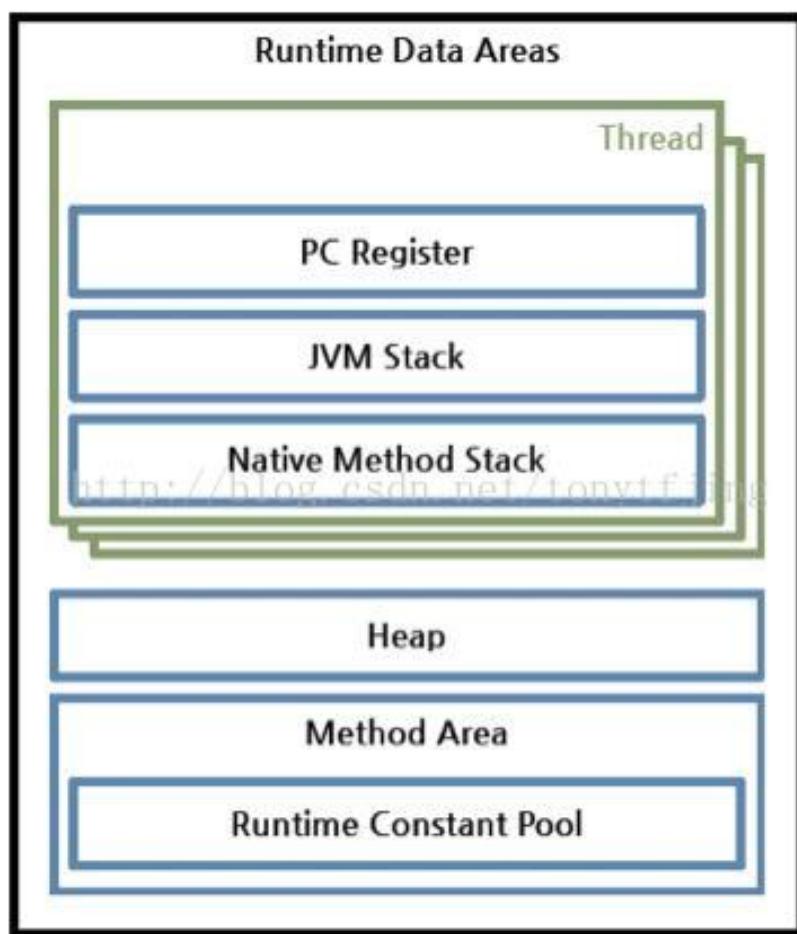


图 5-4 jvmRunTime

5.1.1.8 永久代PermGen 和元空间Metaspace 区别

1. 永久代PermGen : 是jdk1.7 对于方发区的实现. 由于动态生成类的情况比较容易出现永久代的内存溢出, 抛出异常. 而且字符串存储在永久代中容易出现性能问题和内存溢出.
2. 元空间MetaSpace: 存在于本地内存.

5.1.1.9 说一下堆栈的区别?

物理地址

堆的物理地址分配对对象是不连续的. 因此, 性能慢些. 在GC的时候也要考虑到不连续的分配, 所以后各种算法. 比如, 标记-清除, 复制, 标记压缩, 分代(即新生代生活复制算法, 老年代使用标记压缩算法);

栈使用的是数据结构中的栈, 先进后出的原则, 物理地址分配是连续的. 所以性能快.

内存区别

堆因为是不连续的, 所以分配的内存是在运行期确认的, 因此大小不固定. 一般堆大小远大于栈.

栈是连续的, 所以分配的内存大小要在编译器就确认, 大小是固定的.

程序的可见度

堆对于整个应用程序都是共享, 可见的. 栈只对于线程是可见的. 所以也是线程私有. 他的生命周期和线程相同. TIPS:

1. 静态变量放在方法区.
2. 静态的对象还是放在堆.

5.1.1.10 常见的垃圾收集器?

(1) Serial收集器, 单线程收集器, 会stop the word.

(2) ParNew(Parallel Old)收集器, 是Serial收集器的多线程版本. 然后, 并行收集垃圾工作, 此时用户线程也是停止的状态

(3) Parallel Scavenge 收集器(新生代)

多线程收集器, 同样是针对新生代. 停顿时间较短

(4) Serial Old 收集器(老年代)

使用标记整理算法收集老年代垃圾, 单线程.

(5) Parallel old 收集器(老年代)

标记整理算法, 多线程

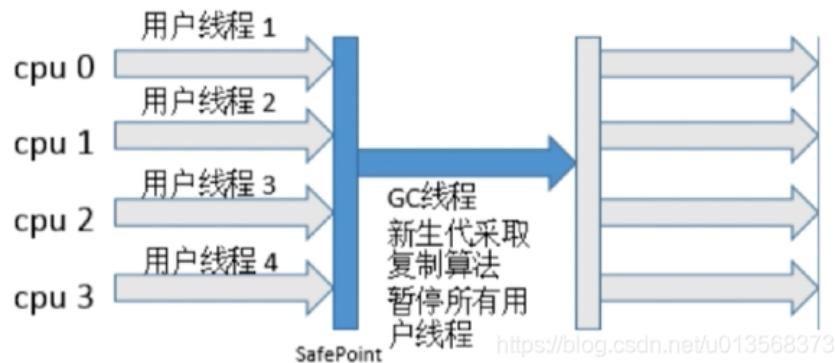


图 5–5 Serial

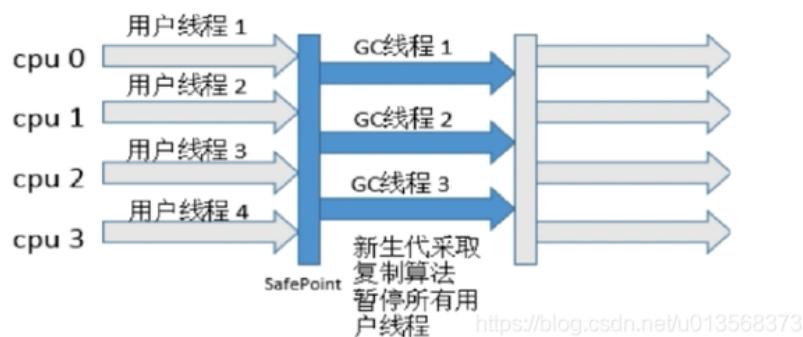


图 5–6 ParNew

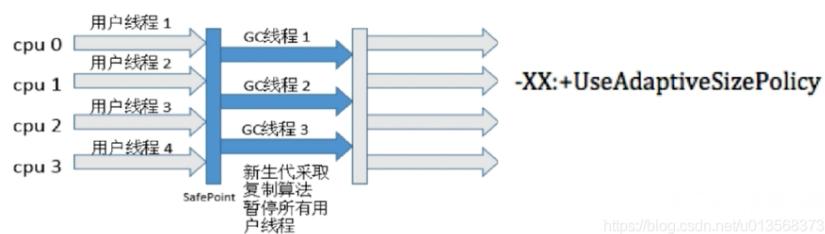


图 5–7 parallelscavenge

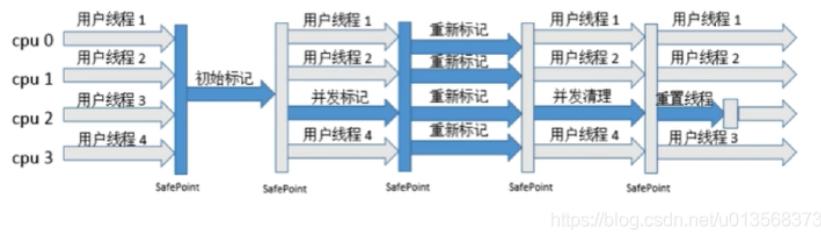


图 5-8 cms

(6) CMS收集器

简单来说,在垃圾回收线程几乎能做到与用户线程同时工作,使用标记清除算法.(7)
G1 收集器

使用复制 + 标记 - 整理算法收集新生代和老年代垃圾.

5.1.1.11 内存分配与回收策略.

1. MinorGC 和 Full GC 有什么不同?

MINORGC: 新生代垃圾回收, 回收速度一般较快

MajorGC: 老年代GC, 回收速度较慢

FULLGC: 重GC, 会清理整个空间包括年轻代和老年代.

2. 什么时候对象进入老年代

(1) 大对象直接进入老年代

(2) 空间分配担保: 当TO被填满后当其中的对象还剩或者, 剩下的对象直接存入老年代

(3) 年龄判定: 如果Survivor空间中相同年龄多有对象大小的总和大于Survivor空间的一半, 年龄大于或等于改年龄的对象就可以直接进入老年代, 如需达到要求的年龄

5.1.1.12 虚拟机性能监控和故障处理工具

jvisualvm 可视化监控.

5.1.1.13 简述JVM中类加载机制

类加载过程: 加载, 验证, 准备, 解析和初始化.

(1) 加载

1. 通过类的全限定名获取此类的二进制字节流

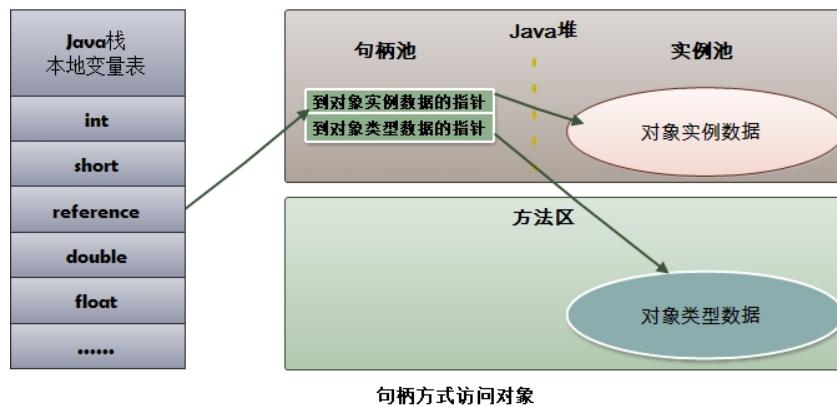


图 5-9 jvmhandler

2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
3. 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口

(2) 验证

为了确保Class文件的字节流中包含的信息符合当前的虚拟机的要求，并且不会危害虚拟机自身的安全。

(3) 准备

正式为类变量(static修饰的)分配内存并设置类变量初始值的节点，这些变量所使用的内存都将在方法区中进行分配

(4) 解析

虚拟机将常量池内的符号引用替换为直接引用的过程。主要对类接口、字段、类方法、接口方法的解析，主要是静态链接，方法主要是静态方法和私有方法。

5.1.1.14 对象的访问定位？

目前主流的访问方式有句柄和直接指针两种方式。

1. 指针：指向对象，代表一个对象在内存中的起始地址
2. 句柄：可以理解为指向指针的指针，维护者对象的地址。句柄不直接指向对象，而是指向对象的地址（句柄不发生变化，指向固定内存地址），再由对象的指针指向对象的真实内存地址。

句柄访问

Java堆中划分出一块内存作为句柄池，引用中存储对象的句柄地址，而句柄中包含了对象实例数据与对象类型数据各自的地址信息，具体构造如下图所示：直接指针

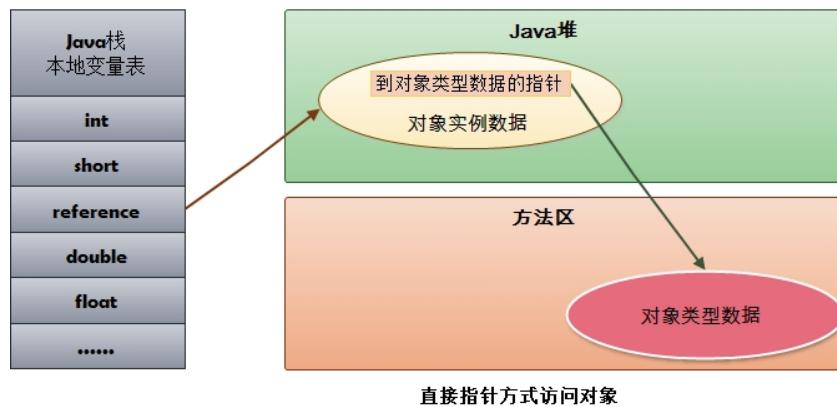


图 5-10 jvmpoint

5.1.1.15 垃圾回收器的基本原理是什么？

可达性分析

GC采用有向图的方式记录和管理堆中的所有对象。通过这种方式确定哪些对象是"可达的",哪些对象是"不可达的",当GC确定一些对象为"不可达"时,GC就有责任回收这些内存空间。程序员可以手动执行System.gc(),通知GC运行,但是Java语言规范并不保证GC一定会执行。

引用计数法 为每个对象创建一个引用计数器,有对象引用时计数器+1,引用被释放时-1,当计数器为0时就可以被回收。优缺点,不能解决循环引用的问题。

5.1.1.16 在java中,对象什么时候可以被垃圾回收?

当对象变的不可触及的时候,这个对象就可以被回收了,垃圾回收不会发生在永久代,如果永久代满了或者是超过了临界值,会触发完全垃圾回收(full gc),会导致Stop-the-world.

5.1.1.17 如何判断对象已经死亡?

- (1) 引用计数法,标记为零。缺点难以解决互相引用问题
- (2) 可达性分析,当一个对象到GC Root对象没有任何路径可达。
- (3) 上面两种都是暂时处于缓刑阶段,真正宣告一个对象死亡,至少要经历两次标记过程。

5.1.1.18 简述强, 软, 弱, 虚引用?

(1) 强引用

如果一个对象具有强引用, 垃圾回收期绝不会回收它

(2) 软引用(SoftRef)

如果内存足够, 垃圾回收期就不会回收它, 如果内存不足了, 就会回收这些对象的内存. 可以实现内存敏感的告诉缓存.

(3) 弱引用(WeakRef)

弱引用关联的对象只能生存到下一次垃圾回收之前.

(4) 虚引用(ReferenceQue)

如果一个对象仅持有虚引用, 那么他就和没有任何引用一样, 在任何时候都可能被垃圾回收. 必须和引用队列一起联合使用.

区别:

软弱引用都是发生在垃圾回收动作之后, 虚引用发生在垃圾回收动作之前.

6

RocketMq

► ...

6.1 知识点和方法论

6.1.1 知识点

NameServer：主要负责对于源数据的管理，包括了对于Topic和路由信息的管理。

每个 Broker 在启动的时候会到 NameServer 注册， Producer 在发送消息前会根据 Topic 到 NameServer 获取到 Broker 的路由信息， Consumer 也会定时获取 Topic 的路由信息。

6.1.1.1 信息流

生产阶段，Producer 新建消息，然后通过网络将消息投递给 MQ Broker 存储阶段，消息将会存储在 Broker 端磁盘中消息阶段， Consumer 将会从 Broker 拉取消息

生产阶段

生产者（Producer）通过网络发送消息给 Broker，当 Broker 收到之后，将会返回确认响应信息给 Producer。所以生产者只要接收到返回的确认响应，就代表消息在生产阶段未丢失。

消费者有两种消费模式，一种是DefaultMQPushConsumer和DefaultMQPullConsumer模式，一是推送消息，一个是拉取消息。

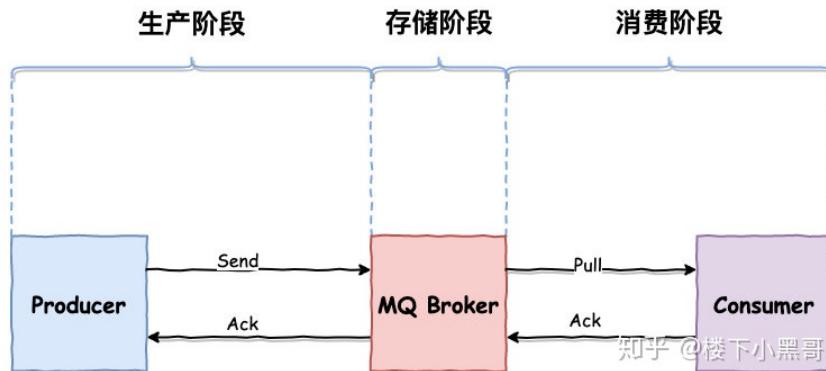


图 6-1 rocketmqlife

push: consumer向broker发出请求, 保持了一种长连接, broker会每5s检测一次是否有消息, 如果有消息, 则将消息推送给consumer. 使用DefaultMQPushConsumer实现消息消费, broker会主动记录消息消费的偏移量

pull: 是消费方主动去broker拉取诗句, 一般会在本地使用定时任务实现, 使用它获得消息状态方便, 负载均衡心梗可控, 但消息的及时性差, 而且需要手动记录消息消费的偏移量, 所以在工作中多数情况推荐使用Push模式.

Broker 存储阶段

将会优先保存到内存中, 然后立刻返回确认响应给生产者。随后 Broker 定期批量的将一组消息从内存异步刷入磁盘。

6.1.1.2 消息队列如何保证消息可靠传输

1. 保证消息不重复. 生产者不能重复生产消息
2. 消息不能少, 消息不丢失, 生产者发送的消息, 消费者一定要能消费到, 对于这个问题
 - a. 生产者发送消息时, 要确认broker确实收到并持久化了这条消息. ack机制.
 - b. broker要等待消费者真正确认消费到了消息时才删除掉消息, 通过消费端的ack机制.

消息实现的时候, 需要一定的幂等性操作. 如果有两条重复的消息发送过来了, 在数据库层面需要做好操作比如插入订单号, 如果已经有了就把这个订单号丢弃.

6.1.1.3 消息队列有哪些作用

1. 解耦: 两个系统的直接通讯方式, 两个系统不需要相互依赖
2. 异步: 系统A给消费队列发送完消息之后, 就可以继续做其他事情了

3. 流量削峰: 如果使用消息队列的方式来调用某个系统, 那么消息将在队列中排队, 由消费者自己控制消费速度.

6.1.1.4 RocketMQ架构设计

nameserver: 独立, 不进行通信, 维护路由信息, 存储了发送者是谁, 消费者是谁, broker是谁

broker: 和每一个nameserver建立一个长连接.

- topic:

— queue: 和消费者负载均衡

producer: 拉取topic所属的broker,

CommitLog: 消息内容, 不区分topic

ConsumeQueue: 基于topic的索引文件

IndexFile: 通过key或时间区间.

6.1.1.5 RocketMQ事物型消息

依赖TransactionListener接口

executeLocalTransaction 方法会在发送消息后调用, 用于执行本地事物, 如果本地事物执行成功, rocketmq再提交消息.

简单来说, 两阶段提交, 消息先提交到RMS_SYS_TRANS_HALF_TOPIC的topic中, 而不是投递到真正的topic中. 当commit成功之后, 将消息投递到真实的topic中, 然后把先提交的地方删除, 如果是rollback只把第一次提交到的地方删除.

7

Redis

► ...

7.1 知识点和方法论

7.1.0.1 布隆过滤器

原理：简单的说就是，有一个长的二进制数组，然后，通过多个hash函数映射到上面，如果某个key没有映射到已经被映射的数组bit位上我们就认为这个数不存在。如果已经映射了，我们就认为这个数存在。但是这有一定的概率失败。如果hash函数都映射到已经映射的位置上，所以有一定的误判率。

可以简单使用google的布隆过滤器，简单的说就是在某些场景下，hashmap的性能消耗过多。不适合做。可以简单解决缓存穿透的问题。即，不存在的对象进行数据库访问我们可以用布隆过滤器进行拦截。

7.1.0.2 如何解决并发数据一致性问题？

1. 使用分布式锁
2. 使用mysql悲观锁 for update, 行锁，加上排它锁，这样后来的事物会阻塞查询，这样就比变了数据不一致。
3. 使用mysql乐观锁，基于CAS 当我心中预想的金额或者版本等于我想要的版本的时候更新整个账户金额

```
Update t_account set money=#new_money where id=#id and money=#old_money;
```

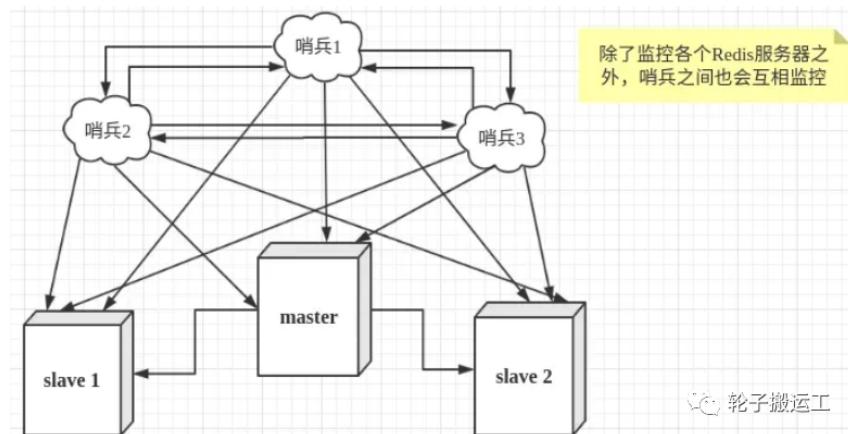


图 7-1 redissentinel

7.1.0.3 Redis的数据结构及使用场景

字符串(string), 用来缓存简单的数据结构, 简单的字符串, 可以实现Session共享
列表(list), Redis的列表通过命令的组合, 即可以当做栈, 也可以当做队列来使用
集合(set), 可以实现自己和某人共同关注的人

散列表(hash), 存储一些key-value对, 更适合用来存储对象

有序集合(sorted set). 设置顺序, 实现排行榜的功能

7.1.0.4 Redis集群策略

一主多从, 整个集群所能存储的数据收到某台机器的内存容量, 所以不可能支持特大数据量, 一般加上哨兵模式来使用

优点:

哨兵模式是基于主从模式的, 所有主从的优点, 哨兵模式都具有。当某个master服务下线时, 自动将该master下的某个从服务升级为master服务替代已下线的master服务继续处理请求sentinel与redis1和redis2建立长连接, 与主机连接是心跳机制,

主从可以自动切换, 系统更健壮, 可用性更高。

缺点:

Redis较难支持在线扩容, 在集群容量达到上限时在线扩容会变得很复杂。

Cluster模式: 槽分配. 支持多主多从为了保证高可用, redis-cluster集群引入了主从模式, 一个主节点对应一个或者多个从节点, 当主节点宕机的时候, 就会启用从节点。当其它主节点ping一个主节点A时, 如果半数以上的主节点与A通信超时, 那么认为主节点A宕机了。如果主节点A和它的从节点A1都宕机了, 那么该集群就无法再提供服务

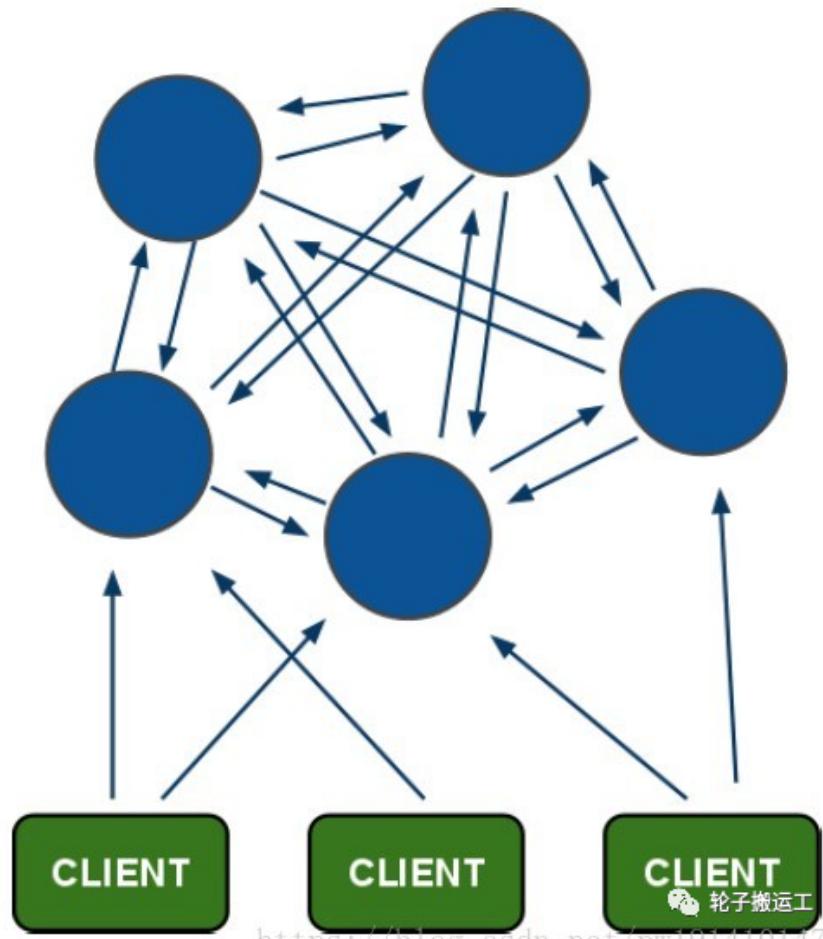


图 7-2 rediscluster

了。

当竞选从节点收到过半主节点同意，便会成为新的主节点。此时会以最新的Epoch通过PONG消息广播，让Redis Cluster的其他节点尽快的更新集群信息。当原主节点恢复加入后会降级为从节点。https://blog.csdn.net/zhibo_lv/article/details/105239297

7.1.0.5 什么是Redis?

1. 高性能非关系型数据库
2. 可以存储五种不同类型的键值之间的映射。键的类型只能为字符串，值支持五种类型数据：字符串(string)，列表(list)，集合(set)，散列表(hash)，有序集合(sorted set)。
3. redis数据是存在内存中的，所以读写速度非常快。

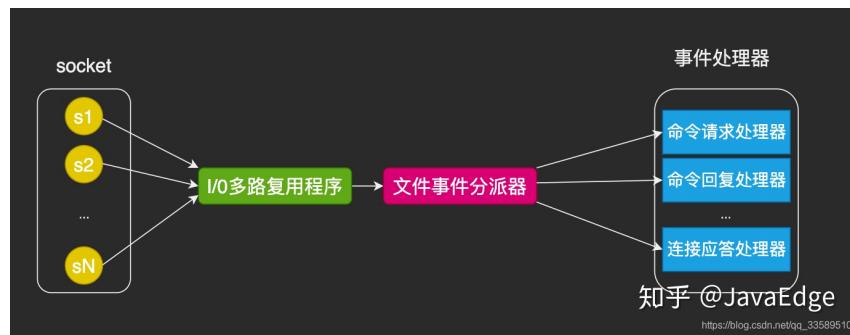


图 7-3 redisfileevent

7.1.0.6 简述Redis单线程模型?

实现方式

(1) I/O多路复用

简单来说, 可以使用I/O多路复用来坚定多个socket连接, 然后将感兴趣的时间注册到内核中并监听每个事件是否发生.

(2) 基于事件驱动

服务器需要处理两类事件, 文件事件; 时间事件.

当被监听的套接字准备好执行连接应答(accept), 读取(read), 写入(write), 关闭(close)等操作时, 与操作相对应的文件事件就会产生, 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件.

文件事件处理器(file event handler) 主要包含4个部分: 多个socket(客户端链接), IO多路复用, 文件事件分派; 事件处理;

7.1.0.7 Redis五种类型数据的实现方式

字符串结构SDS和C中char[]有什么不同

1. 获取SDS中字符串的长度因为SDS中存储了字符串的长度len属性, 直接访问, 时间复杂度O(1), 对于C语言获取字符串的长度需要经过遍历, 时间复杂度O(n).
2. 避免缓冲区溢出, 会检查SDS中属性, free(空闲空间)能够实现字符串的扩充判断. 不足会重新申请空间.
3. SDS支持空间预分配, 扩展的内存比实际需要的多
4. SDS支持空间惰性释放, 字符串缩短之后, 不立即进行空间回收操作. SDS也提供相应API, 可以对冗余空间进行回收.
5. 可以存储二进制, 因为SDS不以回车符号进行终止的判定.

表 7-1 实现

类型	编码
STRING	INT(整形, 在String中存储整形会是的)
STRING	EMBSTR(简单动态字符串, 对于短小的string(44位字符)会使用这种结构)
STRING	RAW(简单动态字符串, 对于稍微长一点的string会使用(44位字符)这种结构)
LIST	QUICKLIST(快表)
LIST	LINKEDLIST(快表)
SET	INTSET(整数集合)
SET	HT(哈希表)
ZSET	ZIPLIST(压缩列表)
ZSET	SKIPLIST(跳表)
HASH	ZIPLIST(压缩列表)
HASH	HT(哈希表)

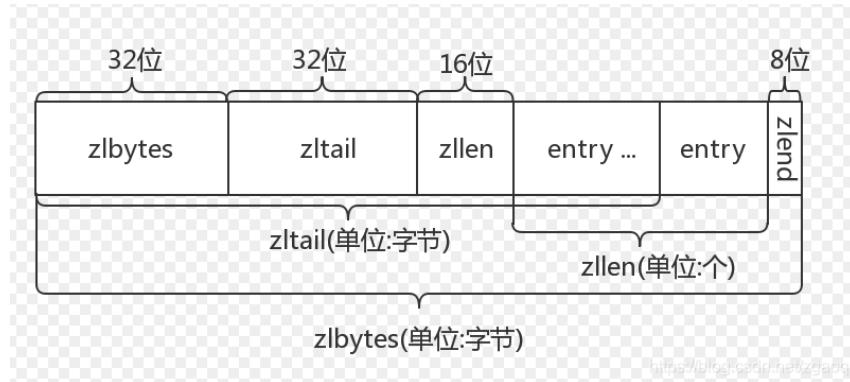


图 7-4 ziplist

7.1.0.8 redis字典的底层实现hashTable相关问题

- 解决冲突: 链地址法, 即使用数组+链表的方式实现.
- 扩容: 有两个指针h[0] 和 h[1], h[1] 用来备份, 当h[0], 满了使用渐进值hash, 插入都插入h[1], 查找两个表都进行查找.

7.1.0.9 压缩链表原理ziplist

连续内存, 包含多个节点entry.

7.1.0.10 zset

本质是枢机链表并有序. skipList与平衡树, 哈希表的比较

1. skipList和各种平衡树的元素排序是有序的, 而哈希表不是有序的, 因此, 在哈希表上智能做单个key的查找, 不适宜做范围查找.
2. 在做范围查找的时候, 平衡树比skipList操作要复杂. 在平衡树上, 需要做一步回退操作. 而在skipList上进行范围查找就非常简单, 只要找到最小值之后对第一层链表进行若干部遍历就可以实现.
3. 平衡树插入和删除操作, 会引起结构调整, 操作复杂, skipList的插入和删除只需要修改相邻节点的指针, 操作简单又快速.

7.1.0.11 AOF和RDB两种持久化方式区别

1. AOF存储命令, RDB存储数据.
2. AOF文件因为存储命令, 所以在redis启动的时候加载aof会比加载rdb要慢.
3. Redis 4.0 之后启动了混合模式, AOF不需要是全量日志, 只要保存前一次RDB存储开始到这段时间增量AOF日志即可.

7.1.0.12 Redis中过期策略和缓存淘汰机制

1. 定期删除: redis默认每隔100ms随机对key检查, 有过期的key则进行删除. 容易导致很多过期的key没被发现
2. 惰性删除: 获取某个key的时候, redis会检查一下, 如果过期了就进行删除.

7.1.0.13 为什么要使用Redis

1. 高性能: 内存的读取比硬盘乃至固态硬盘的读取速度都要快得多
2. 高并发: 直接操作缓存能够承受的请求是远远大于直接访问数据库的, 所以我们可以考虑把数据库中的部分数据转移到缓存中去, 这样用户的一部分请求会直接请求缓存这里而不用经过数据库.
3. 高性能: 使用多路I/O复用模型, 非阻塞IO;

7.1.0.14 Redis底层实现跳表介绍一下

跳表是带多级索引的链表, 时间复杂度 $O(\lg n)$, 所能实现的功能和红黑树差不多, 但是跳表有一个区间查找的优势, 红黑树没有.

1. 表头: 负责维护跳表的节点指针.

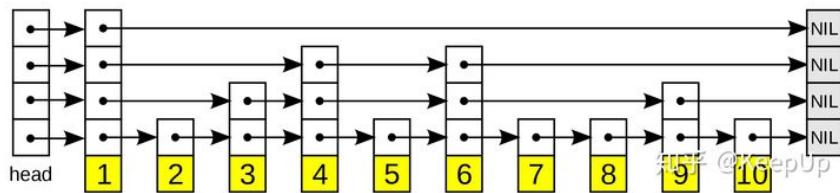


图 7-5 jumptable

2. 跳跃表节点: 保存着元素值, 以及多个层.
3. 层: 保存着指向其他元素的指针, 高层的指针越过的元素数量大于等于底层的指针, 为了提高查找效率, 程序总是从高层先开始访问, 然后随着元素值范围的缩小, 慢慢降低层次.
4. 表尾: 全部由NULL组成.

7.1.0.15 为什么要使用Redis而不用map/guavaCache做缓存

1. guavaCache实现的是本地缓存, 最主要的特点是轻量化以及快速, 生命周期随着jvm的销毁而结束, 并且在多实例的情况下, 每个实例都需要个字保存一份缓存, 缓存容易出现不一致性.
2. 使用redis之类的缓存称为分布式缓存, 在多实例的情况下, 各实例公用一份缓存数据, 缓存具有一致性.

7.1.0.16 分布式锁如何使用redis实现

1. setnx命令原子性实现.

7.1.0.17 Redis的内存淘汰策略有哪些

Redis的内存淘汰策略是指在Redis用于缓存的内存不足时, 怎么处理需要新写入且需要申额外空间的数据. 全局的键空间选择性移除

1. noeviction: 当内存不足以容纳新写入数据时, 新写入操作会报错.
2. allkeys-lru: 当内存不足以容纳新写入数据时, 在键空间中, 移除最近最少使用的key.
3. allkeys-random: 当内存不足以容纳新写入数据时, 在键空间随机移除某个key.

设置过期时间的键空间选择性移除

1. volatile-lru: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间中, 移除最近最少使用的key.

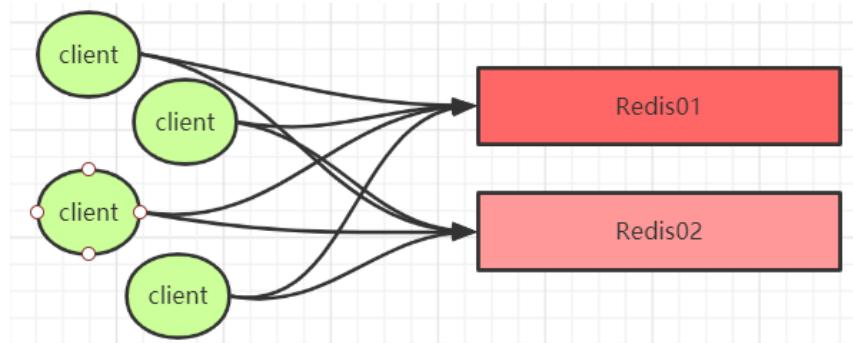


图 7-6 redis_sharding

2. volatile-random: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间中, 随机移除某个key.
3. volatile-ttl: 当内存不足以容纳新写入的数据时, 在设置了过期时间的键空间中, 有更早过期时间的key优先移除.

7.1.0.18 Redis事物的概念

Redis事物的本质通过MULTI, EXEC, WATCH等一组命令的集合.

1. 事物开始 MULTI
2. 命令入队
3. 事务执行 EXEC

* 事务执行过程中, 如果服务端收到有EXEC, DISCARD, WATCH, MULTI之外的请求, 将会把请求放入队列中排队. 简单介绍一下watch, 当watch的变量在事务过程中发生了改变, 那么事务失败, 拒绝执行事物.

```

1 >watch 'name'
2 >multi
3 >set "name" "peter"
4 >exec
5 (nil)
  
```

7.1.0.19 RedisSharding问题

简单来说就是多个client 连接多个redis, 然后通过一致性哈希算法, 来确定访问的key+访问的客户端名字在哪一台redis上. 采用的算法是MURMUR_HASH,

7.1.0.20 缓存雪崩

缓存同一时间大面积的失效, 所以, 后面的请求都会落到数据库上, 造成数据库短时间内承受大量请求而崩掉

解决方案

1. 缓存数据的过期时间设置随机, 防止同一时间大量数据过期现象发生.

7.1.0.21 缓存穿透

缓存和数据库中都没有数据, 导致所有的请求都落在数据库中, 造成数据库短时间内承受大量请求而崩掉.

解决方案

1. 从缓存中取不到的数据, 在数据库中也没有取到, 这时也可以将key-value对写为key-null, 缓存时间设定为30s.

7.1.0.22 缓存击穿

缓存中没有但数据库中有的数据, 由于并发用户特别多, 同时读缓存没读到数据, 又同时去数据库取数据, 引起数据库压力瞬间增大, 造成过大压力, 和缓存雪崩不同的是, 缓存击穿指并发查询同一条数据, 缓存雪崩是不同数据都过期了, 很多数据都查不到, 从而查数据库.

解决方案

1. 设置热点数据永不过期

7.1.0.23 缓存预热

系统上线后, 将相关的缓存数据直接加载到缓存系统. 这样就可以避免在用户请求的时候, 先查询数据库, 然后再讲数据缓存的问题, 用户直接查询事先被预热的缓存数据.

解决方案

1. 定时刷新缓存;

7.1.0.24 Redis6.0 为什么要引入多线程呢?

Redis将所有数据放在内存中, 内存的响应时间大约100ns, 对于小数据包, Redis服务器可以处理8w到10wQPS, 这也是Redis处理的极限了, 对于80%的公司来说, 单线程的redis已经足够使用了.

但随着越来越复杂的业务场景, 需要更大的QPS.



图 7-7 redismasterslave

1. 可以充分利用服务器CPU资源, 目前主线程只能利用一个核
2. 多线程任务可以分摊Redis同步IO读写负荷.

7.1.0.25 Redis主从复制模式

1. 完全同步:刚开始, 主服务器发送RDB文件给从服务器, 实现主从同步.
2. 部分同步:当连接由于网络原因断开的时候, 将中间断开的执行的写命令发送给从服务器. 实现同步.

简单来说, 从服务器发送SYNC信号给主服务器, 主服务将RDB快照发送给从服务器, 从服务器更新快照内容. 主服务器将缓存写命令发送给从服务器.

7.1.0.26 Redis中持久化机制

- 1.

8

计算机网络

► ...

8.1 知识点和方法论

8.1.0.1 查看端口信息

netstat -a // 显示所有的端口信息

8.1.0.2 SYN攻击

简单的说是三次握手环节, 攻击者发送SYN请求, 然后就消失了, 导致服务器上大量的资源尝试去进行SYN响应.

解决办法, 降低SYN timeout时间, 使得主机尽快释放半连接的占用

8.1.0.3 arp协议

根据IP地址查询相应的以太网MAC地址

8.1.0.4 DNS过程

1. 先从DNS缓存中, 本地缓存或者路由器缓存中查找ip地址. DNS根服务器, .com服务器, .server.com 服务器三级查找

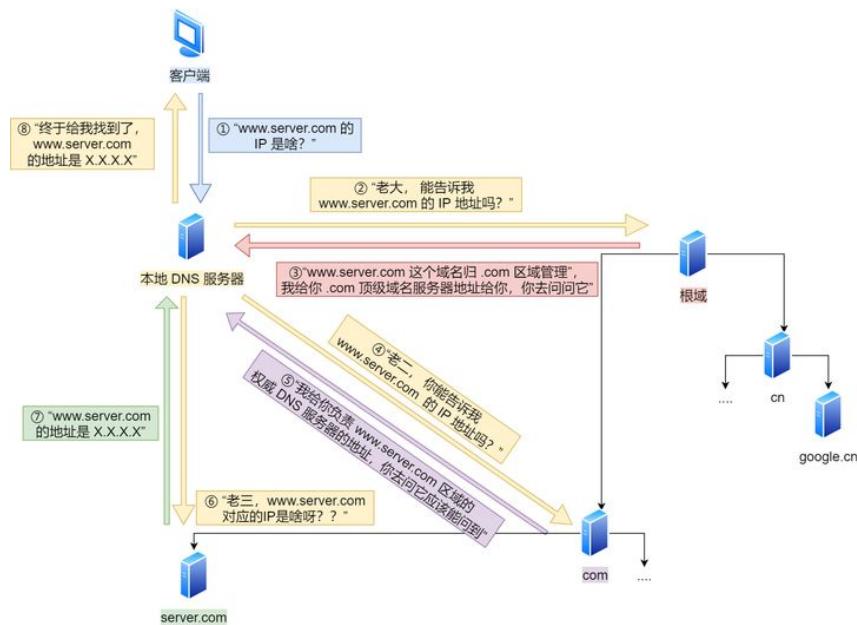


图 8-1 dns

8.1.0.5 TCP如何保证可靠性

1. 校验和: 通过添加16byte的校验和
2. 确认信号ACK与序列号数据发送后, 对方回发送一个ack表示数据已经收到了比如服务端发送了1-1000的数据, ack应答1001 表示前1000个数据已经送达了.
3. 超时重传: 报文发出后在一定时间内未收到接收方的应答, 发送方就会进行重发.
4. 连接管理机制: 三次握手和四次挥手
5. 流量控制: 接收端处理数据的速度是有限的, 如果发送方发送数据的速度过快, 导致接收端的缓冲区满, 而发送方继续发送, 就会造成丢包, 继而引起丢包重传等一系列连锁反应。因此TCP支持根据接收端的处理能力, 来决定发送端的发送速度, 这个机制叫做流量控制。在TCP报文段首部中有一个16位窗口长度, 当接收端接收到发送方的数据后, 在应答报文ACK中就将自身缓冲区的剩余大小, 放入16窗口大小中。这个大小随数据传输情况而变, 窗口越大, 网络吞吐量越高, 而一旦接收方发现自身的缓冲区快满了, 就将窗口设置为更小的值通知发送方。如果缓冲区满, 就将窗口置为0, 发送方收到后就不再发送数据, 但是需要定期发送一个窗口探测数据段, 使接收端把窗口大小告诉发送端。
6. 拥塞控制然而如果网络非常拥堵, 此时再发送数据就会加重网络负担, 那么发送的数据段很可能超过了最大生存时间也没有到达接收方, 就会产生丢包问题。为此TCP引入慢启动机制, 先发出少量数据, 就像探路一样, 先摸清当前的网络拥堵状

态后，再决定按照多大的速度传送数据。此处引入一个拥塞窗口：发送开始时定义拥塞窗口大小为1；每次收到一个ACK应答，拥塞窗口加1；而在每次发送数据时，发送窗口取拥塞窗口与接发送段接收窗口最小者。慢启动：在启动初期以指数增长方式增长；设置一个慢启动的阈值，当以指数增长达到阈值时就停止指数增长，按照线性增长方式增加；线性增长达到网络拥塞时立即“乘法减小”，拥塞窗口置回1，进行新一轮的“慢启动”，同时新一轮的阈值变为原来的一半。慢启动，拥塞避免，快重传，快恢复。

8.1.0.6 TCP报文头部内容

1. 源端口
2. 目的端口
3. 校验和
4. ACK标志位
5. FIN 标志位
6. 校验和
7. 序号seq和确认号ack

8.1.0.7 TCP三次握手和四次挥手示意图

<https://www.cnblogs.com/eat-too-much/p/14764778.html> <https://www.cnblogs.com/eat-too-much/p/14765125.html>

8.1.0.8 Time_wait状态

1. 一个TCP连接中，主动关闭连接的一方发现的状态；收到FIN命令，进入TIME_WAIT状态，并返回ACK命令

2. 保持两个MSL时间，即4分钟；(MSL为2分钟)

如果在高并发场景下，出现了大量的tcpTimewait状态。如何解决？因为tcp头中有16bit表示端口号。如果65536个端口号因为timewait状态被占用，系统就无法给出可用的服务。

缩减time_wait时间，设置为1msl。

服务端中可以设置为不必等待2MSL时间结束，即可使用被占用的端口

8.1.0.9 SYN-RCVD状态

简单来说，就是三次建立连接的时候，对方没有把最后一次ack发送回来的时候的状

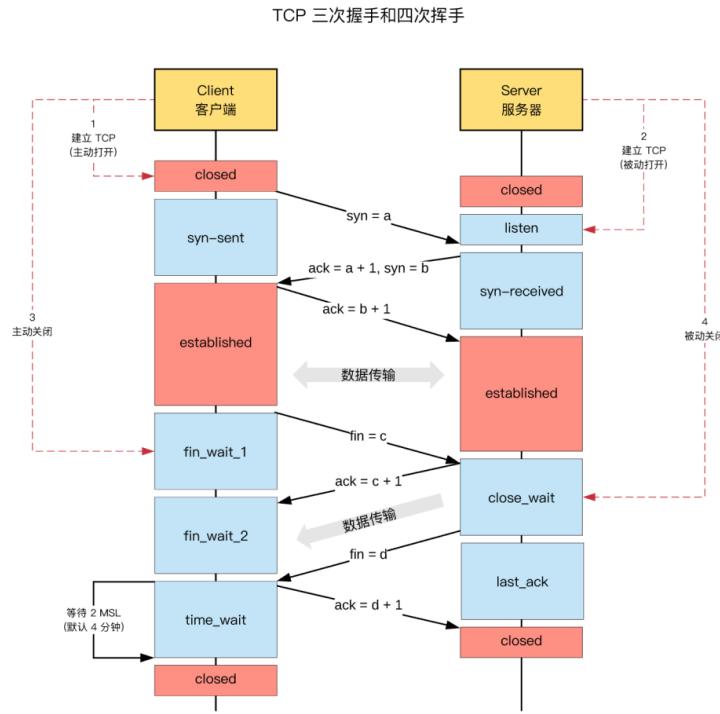


图 8-2 tcpconnectdis

态. 可能是别人攻击服务器产生的.

开启cookie检查校验连接的合法性

`net.ipv4.tcp_synccookies = 1`

8.1.0.10 TCP四个计时器

1、重传计时器

一段时间没有接收到应答,则开始重新传输数据, 约60s

2. FIN 时间等待计时器

时间等待计时器是在连接终止期间使用的。当TCP关闭一个连接时, 它并不认为这个连接马上就真正地关闭了。在时间等待期间中, 连接还处于一种中间过渡状态。这就可以使重复的FIN报文段(如果有的话)可以到达目的站因而可将其丢弃。这个计时器的值通常设置为一个报文段的寿命期待值的两倍。

3. 保活计时器

保活计时器通常设置为2小时。若服务器过了2小时还没有收到客户的信息, 它就发送探测报文段。若发送了10个探测报文段(每一个相隔75秒)还没有响应, 就假定客户出了故障, 因而就终止该连接。

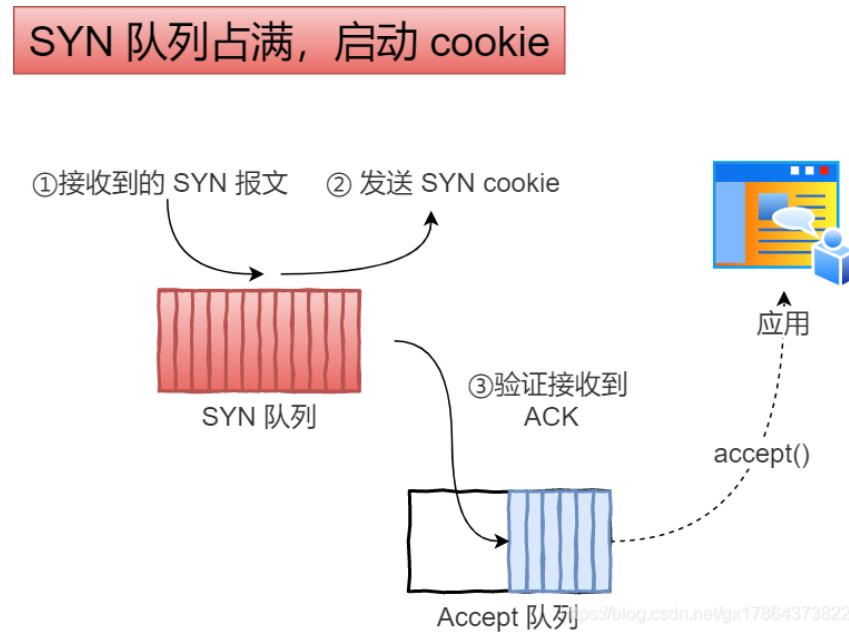


图 8-3 tcpsynrcvd

4、坚持计时器

当发送TCP收到一个窗口大小为零的确认时，就启动坚持计时器。当坚持计时器期限到时，发送TCP就发送一个特殊的报文段，叫做探测报文段。这个报文段只有一个字节的数据。它有一个序号，但它的序号永远不需要确认；甚至在计算对其他部分的数据的确认时该序号也被忽略。探测报文段提醒对端：确认已丢失，必须重传。

8.1.0.11 TCP流量控制

简单来说，接收端通过返回一个win参数告诉发送端还能发送多少。如果不能发送了的话，开启坚持定时器，去发送试探帧，如果又可以发送了的话，继续发送。

新一代算法采用google的BBR算法。

TCP BBR 不再使用丢包作为拥塞的信号，也不使用“加性增，乘性减”来维护发送窗口大小，而是分别估计极大带宽和极小延迟，把它们的乘积作为发送窗口大小。

8.1.0.12 HTTPS访问过程, SSL 握手的过程

证书主要作用是在SSL握手中，我们来看一下SSL的握手过程

1. 客户端提交https请求
2. 服务器响应客户，并把证书公钥发给客户端
3. 客户端验证证书公钥的有效性

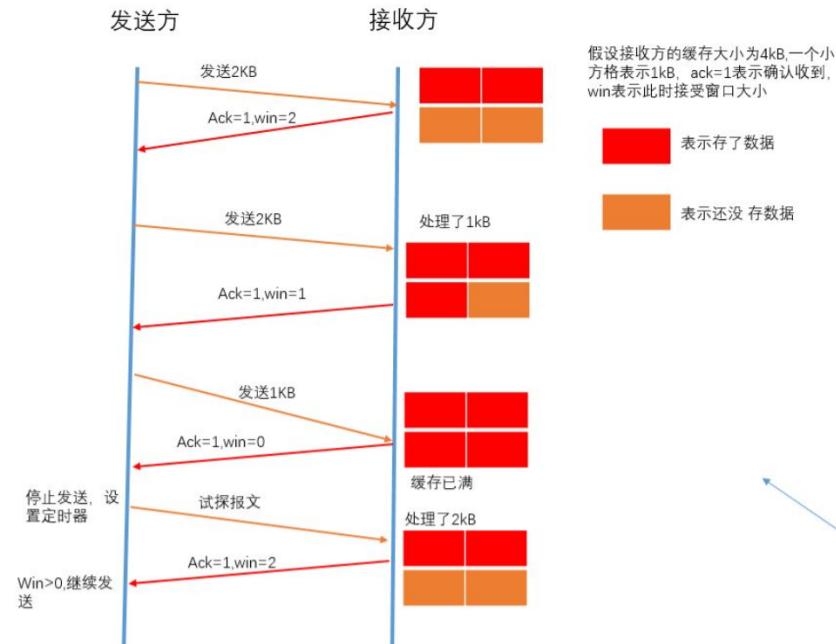


图 8-4 liuliangkongzhi

4. 有效后，会生成一个会话密钥
5. 用证书公钥加密这个会话密钥后，发送给服务器
6. 服务器收到公钥加密的会话密钥后，用私钥解密，回去会话密钥
7. 客户端与服务器双方利用这个会话密钥加密要传输的数据进行通信

8.1.0.13 计算机网络分层

计算机网络OSI模型是七层, 如果是TCP/IP则是四层

8.1.0.14 TCP和UDP区别?

UDP: 面向报文, 支持1对1, 1对多, 多对1的交互通信

TCP: 面向连接, 提供可靠交付, 有流量控制, 拥塞控制, 提供全双工通信, 面向字节流. TCP是点对点的.

8.1.0.15 TCP三次握手相关问题

为什么三次握手而不是两次握手: 主要是为了防止已失效的链接请求报文段突然又传送到了B, 因而产生错误. 如果只要两次握手就建立连接, 那么如果A第一次请求丢

RSA密钥交换

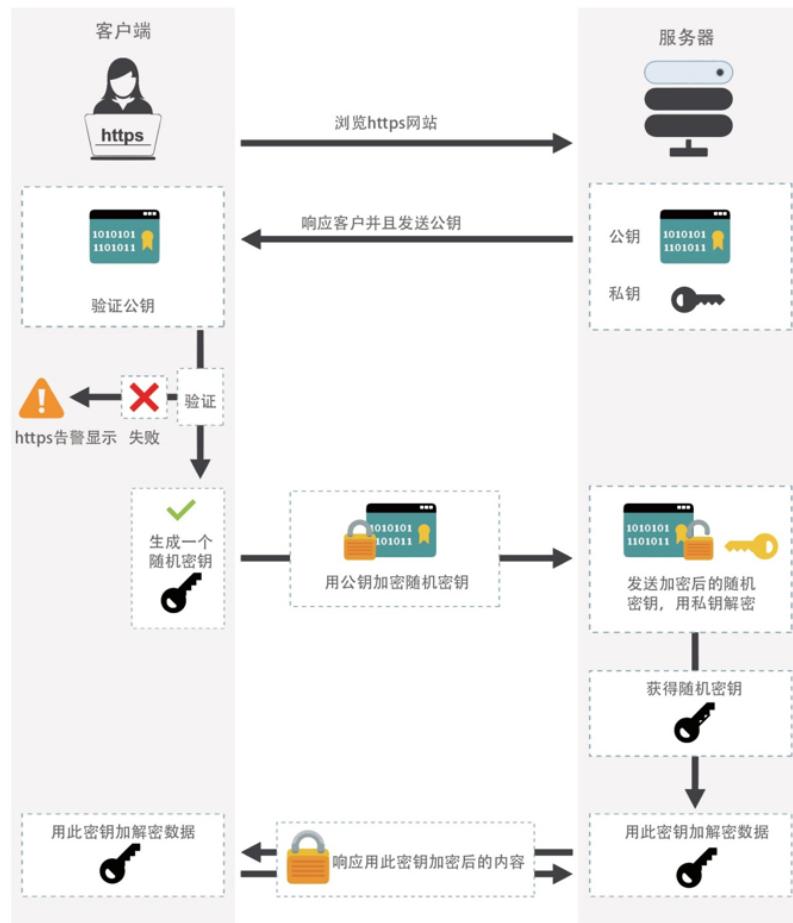


图 8-5 https

OSI 体系结构	TCP/IP 协议集	
应用层		
表示层	应用层	TELNET、FTP、HTTP、SMTP、DNS 等
会话层		
传输层	传输层	TCP、UDP
网络层	网络层	IP、ICMP、ARP、RARP
数据链路层		
物理层	网络接口层	各种物理通信网络接口 https://blog.csdn.net/leho666

图 8-6 TCP_IP_OSI

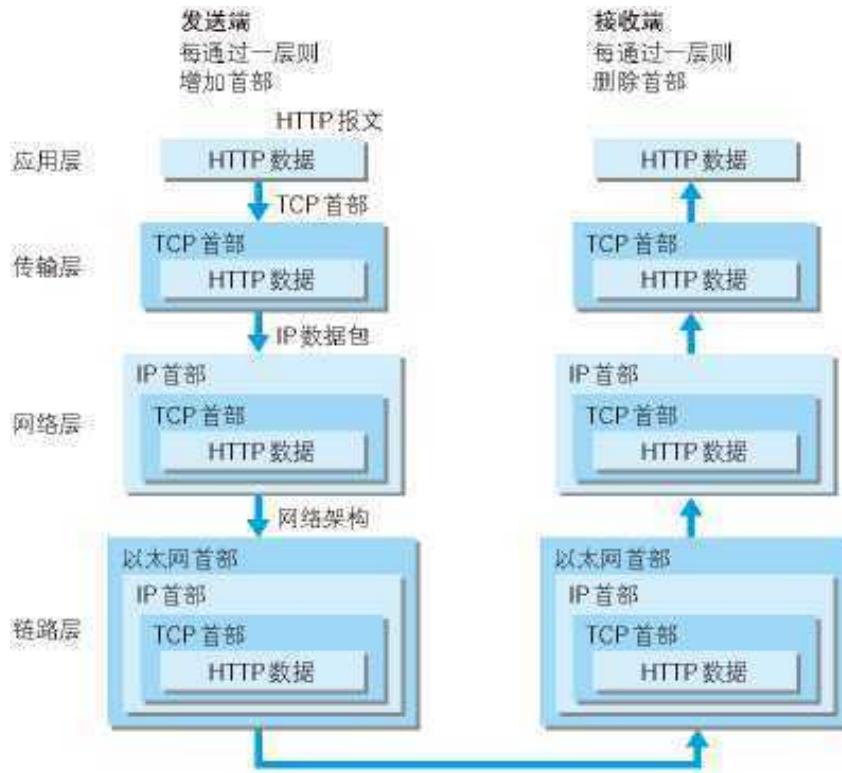


图 8-7 osi

失了, A 又发送了一次请求, 但是这一次传输结束了, A上一次丢失的请求再次被传送到了B, 如果建立了连接, 但是A现在一直不回应B, 导致B浪费了资源.

8.1.0.16 TCP四次挥手问题

关闭连接的时候, 当Server端收到FIN报文时候, 很可能并不会立即关闭SOCKET, 所以只能先回复一个ACK报文, 告诉Client端, 你发送的FIN我收到了, 只有等我Server端所有的报文都发送完了, 我才能发送FIN报文. 不能一起发送, 所以需要四步挥手.

8.1.0.17 TCP协议-如何保证传输的可靠性

超时重传: 简单理解就是发送在发送完数据后等待一个时间, 时间到大没有接收到ACK报文, 那么对刚才发送的数据进行重新发送

连接管理: 使用三次握手和四次挥手

流量控制: TCP根据接收端对数据的处理能力, 决定发送端的发送速度, 这个机制就是流量控制. TCP协议的包头信息当中, 有一个16位字段的窗口大小, 发送方更具ACK报文里的窗口大小的值的改变自己的发送数据

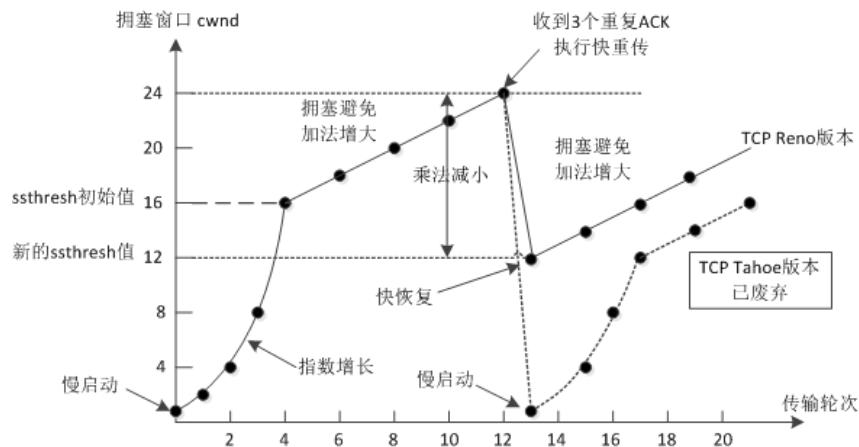


图 8-8 tcp

拥塞控制: 慢开始, 拥塞避免, 快重传, 快恢复.

慢开始算法的思路就是, 不要一开始就发送大量的数据, 先探测一下网络的拥塞程度, 也就是说由小到大逐渐增加拥塞窗口的大小.

拥塞避免算法让拥塞窗口缓慢增长, 即每经过一个往返时间RTT就把发送方的拥塞窗口 cwnd 加1, 而不是加倍. 这样拥塞窗口按线性归路缓慢增长.

快重传和快恢复: 发送方只要一收到三个重复确认就启动立即重传对方尚未收到的报文段, 而不必继续等待设置的重传计时器时间到器.

8.1.0.18 Cookie作用, 安全性问题和Session的比较

(1) Cookie是服务器发送到用户浏览器并保存在本地的一小块数据, 他会在浏览器之后向统一服务器再次发送请求时被携带上.

(2) Session 存储在服务器端. 使用Session维护用户登录状态的过程如下: (需要cookie作为传输机制) 用户进行登录时, 用户提交包含用户名和密码的表单, 放入HTTP请求报文中; 服务器校验该用户名和密码, 如果正确则把用户信息存储到Redis中, 他在Redis中的key成为SessionId.

服务器返回的响应报文Set-Cookie首部字段包含了这个SessionID, 客户端收到响应报文之后将该Cookie值存入浏览器中. 客户端之后对同一个服务器进行请求时会包含该Cookie值, 服务器收到之后提取出SessionID, 从Redis中取出用户信息, 继续之前的业务操作.

session 的运行依赖 session id, 而 session id 是存在 cookie 中的, 也就是, 如果浏览器禁用了 cookie, 同时 session 也会失效(但是可以通过其它方式实现, 比如在 url 中传递 session_id)

cookie存在大小限制, 单个不超过4k, 浏览器中Cookie个数也有限制. Session没有大小限制, 是和服务器内存有关的.

8.1.0.19 HTTP1.1 和 HTTP1.0的比较

HTTP1.1 默认长链接, 长连接只需要建立一次TCP连接, 进行多次HTTP通信.

纯文本协议

HTTP1.0 默认短连接, 每进行一次HTTP通信就要新建一个TCP连接.

8.1.0.20 HTTP2.0 和 HTTP1.1的比较

2.0 特点

多路复用: 所有HTTP2.0通信都在一个TCP链接上完成, 这个链接可以承载任意流量的双向数据流。

二进制分帧: + 头部压缩

服务器推送: 服务器除了最初请求的响应外, 服务器还可以额外向客户端推送资源, 而无需客户端明确的需求。

这种方式如何防止接受混乱呢?

在HTTP2.0上, 客户端和服务器可以把HTTP 消息分解为互不依赖的帧, 然后乱序发送, 最后再在另一端把它们重新组合起来。注意, 同一链接上有多个不同方向的数据流在传输。客户端可以一边乱序发送stream, 也可以一边接收者服务器的响应, 而服务器那端同理。(简单来说就是, 根据请求头来组件一个整个数据)

也会遇到队头阻塞, TCP层面上.

1.1

对头阻塞: 如果4丢了, 那么收到的5-8都没用. 有非管道化和管道化, 两种方式。

非管道化, 完全串行执行, 请求->响应->请求->响应..., 后一个请求必须在前一个响应之后发送。

管道化, 请求可以并行发出, 但是响应必须串行返回。后一个响应必须在前一个响应之后。原因是, 没有序号标明顺序, 只能串行接收。

管道化请求的致命弱点:

1. 会造成队头阻塞, 前一个响应未及时返回, 后面的响应被阻塞 2. 请求必须是幂等请求, 不能修改资源。因为, 意外中断时候, 客户端需要把未收到响应的请求重发, 非幂等请求, 会造成资源破坏。

由于这个原因, 目前大部分浏览器和Web服务器, 都关闭了管道化, 采用非管道化模式。

无论是非管道化还是管道化，都会造成队头阻塞(请求阻塞)。

8.1.0.21 HTTP3.0

QUIC协议:

前项纠错机制: 称为向前纠错(Foward Error Connec, FEC), 每个数据包除了它本身的内容之外还包括了其他数据包的数据, 因此少量的丢包可以通过其他包的冗余数据直接组装而无需重传。

基于UDP的协议.

自定义连接机制:

TCP基于四元数,IP, 目标IP, 目标port, 自己的port. 一旦一个元素发生变化, 就会重新建立连接. 但是, QUIC使用一个64位随机数来确定这个连接. 三次握手的时间减少. 也减少二路TLS的时间.

8.1.0.22 HTTPS加密

使用SSL连接. HTTPS采用混合加密算法, 使用非对称加密和对称加密, 非对称加密用于传输对称秘钥来保证传输过程的安全性, 之后使用对称秘钥加密进行通信来保证通信过程的效率.

HTTPS加密过程:

1. 客户使用HTTPS的URL访问web服务器, 要求与Web服务器建立SSL链接.
2. Web服务器收到客户端请求后, 会将网站的证书信息(证书中包含公钥)传送一份给客户端.
3. 客户端的浏览器与web服务器开始写上SSL链接的安全登记, 也就是信息加密登记.
4. 客户端的浏览器更具双方同意的安全登记, 建立会话秘钥, 然后利用网站的公钥将会话秘钥加密, 并传送给网站
5. Web服务器利用自己的私钥解密出会话秘钥.
6. Web服务器利用会话秘钥加密与客户端之间的通信.

缺点: 因为需要进行加密解密等过程, 因此速度会更慢, 需要支付证书授权的高额费用.

8.1.0.23 输入网址发生的事情

1. 浏览器查找该域名的IP地址

2. 浏览器更具解析得到的IP地址向web服务器发送一个HTTP请求.
3. 服务器收到请求并进行处理
4. 服务器返回一个响应.
5. 浏览器对该响应进行解码, 渲染显示.
6. 页面显示完成后, 浏览器发送异步请求.

9

mysql

► ...

9.1 知识点和方法论

9.1.0.1 索引的使用原则

对于区分度高的索引建议使用索引.

索引可以提高查询速度

索引降低了数据更新操作的速度

9.1.0.2 主键自增

如果输入了一个值比如100, 那么下一个从100快开始自增.

9.1.0.3 主键自增的好处

数据库自动编号, 速度快, 而且是增量增长, 按顺序存放, 对于检索非常有利;
(因为是聚簇索引, 不会造成已经构建好的mysql页进行分裂.) 效率大大提高

9.1.0.4 行锁共享锁

行锁

```
select * from XXX for update;
```

共享锁

```
select * from XXX lock in share mode;
```

9.1.0.5 回表

当你建立了一个索引非主键索引，那么叶子节点存储的是主键索引所在的地方，然后会去主键索引中查找真实数据，这种查询了两个索引就叫回表。

9.1.0.6 mybatis#和\$区别以及原理

可以防止Sql注入，它会将所有传入的参数作为一个字符串来处理。

\$ 则将传入的参数拼接到Sql上去执行，一般用于表名和字段名参数，\$所对应的参数应该由服务器端提供，前端可以用参数进行选择，避免Sql注入的风险

9.1.0.7 redo log 和 undo log, bin log

如果修改直接修改到日志中的话，性能开销比较大，mysql 会将要写的数据先写到redolog中，然后再讲redolog存储到磁盘中，使用场景系统崩溃恢复

但只有 redo log 也不行，因为 redo log 是 InnoDB特有的，且日志上的记录落盘后会被覆盖掉。因此需要 binlog和 redo log二者同时记录，才能保证当数据库发生宕机重启时，数据不会丢失。因为Checkpoint之前的页都已经刷新回磁盘。数据库只需对Checkpoint后的重做日志进行恢复，这样就大大缩短了恢复的时间。

bin log: 归档日志，通过追加方式记录，使用场景：主从复制和数据恢复，简单理解就是二进制sql数据

undo log: 回滚日志，MVCC多版本并发。快照读是MVCC的一种体现方式。比如一条 INSERT 语句，对应一条DELETE 的 undo log.update语句还是update只不过数据是之前的数据，这就是undo log中记录的日志内容。有两个数据，事物id 和回滚指针。如果失败会产生回滚

梳理下事务执行的各个阶段：

- (1) 写undo日志到log buffer；
- (2) 执行事务，并写redo日志到log buffer；
- (3) 如果innodb_flush_log_at_trx_commit=1，则将redo日志写到log file，并刷新落盘。
- (4) 提交事务。

9.1.0.8 隔离级别

读已提交(read committed): 简单来说select * from account, 读出来的数据都是已经提交的数据.

可以出现不可重复读(一个事物两次读到的事物不一样)和幻读(插入一条记录)

读未提交(read uncommitted) : 简单来说 select * from account, 读出来的数据是另一个事物还没有提交的数据,

可以出现脏读(无论是脏写还是脏读, 都是因为一个事务去更新或者查询了另外一个还没提交的事务更新过的数据。因为另外一个事务还没提交, 所以它随时可能会回滚, 那么必然导致你更新的数据就没了, 或者你之前查询到的数据就没了, 这就是脏写和脏读两种场景。): 简单的说, 就是A事物在还没commit数据的情况下, b事物使用了A事物的数据, 但是A事物出现了回滚操作, 导致B事物读出来的数据是脏数据. 可以出现不可重复度, 可以出现幻读. (感觉是最不靠谱的一个级别)

可重复读(RR: Repeatable read) : 简单来说, 就是A事物插入了一条新数据, B事物没看到, 因为可重复读, 但是B是可以操作这条新数据的.

串行化(serializable): 只能一个一个处理, 没什么问题, 但是效率太低一般不考虑.

提示: 不可重复读对应的是修改即Update, 幻读对应的是插入即Insert.

TIPS: 可见, 幻读就是没有读到的记录, 以为不存在, 但其实是可以更新成功的, 并且, 更新成功后, 再次读取, 就出现了。

9.1.0.9 ACID

原子性: 不可分割

一致性: 数据完整性, 例如金钱的综述

隔离性: 不被打扰

持久性: 永久保存

9.1.0.10 乐观锁和悲观锁

悲观锁, 指的是在整个数据处理过程中, 将数据处于锁定状态. 悲观锁的实现, 往往依靠数据库提供的锁机制.

乐观锁的实现, 基于并发控制的CAS理论.

9.1.0.11 MVCC

MVCC用于提交读和可重复读这两种隔离级别. 使用undolog实现

主要依靠事务版本号来实现读已提交和可重复读.

当开始一个新事物时, 该事物的版本号肯定会大于当前所有数据行快照的创建的版本号

9.1.0.12 RR和RC隔离级别下的InnoDB快照读有什么区别

(1) RR隔离级别下, 当事物第一次进行快照读, 仅此一次创建read-view视图, 所以read-view 中未提交事物数组和最大事物ID始终保持不变, 因此每次读取时只会读取事物之前的数据

(2) 而在RC隔离级别下, 每次事物进行快照读是, 都会生成新的read-view视图, 导致在RC隔离级别下事物可以看到其他事物修改后的数据, 这也是导致不可重复的原因

总之, 在RC隔离级别下, 是每个快照读都会生成并获取最新的Read-View: 而在RR隔离级别下, 则是同一个事物中的第一个快照读才会创建Read View, 之后的快照读获取的都是同一个Read View

9.1.0.13 B+/B树之间的比较

(1) B+树空间利用率更高, 可减少IO次数.

(2) B+树所有关键字的查询路径长度相同(因为关键字在叶子节点), 导致每一次查询的效率相当, 更加稳定.

(3) B+叶子节点有指针, 支持between查找很方便.

9.1.0.14 聚集索引&非聚集索引

简单来说, 一个表只有一个聚集索引, 类似于主键索引.

一个表可以有多个非聚集索引, 打比方, 新华字典A-Z的查询事主键索引, 新华字典偏旁查询事非聚集索引. 是会跳跃的.

9.1.0.15 创建索引的优点

(1) 当数据量增大的时候, 索引可以极大的加快数据的查找速度

9.1.0.16 创建索引的缺点

(1) 对于一个表而言, 不单单要维护数据的存储, 也要维护索引的存储. (空间增大)

(2) 对表中的数据进行修改的时候, 索引也要进行动态维护, 这样就降低了数据的维护速度.

9.1.0.17 MYSQL优化

- (1) 最左前缀法则
- (2) 较长的数据列建立前缀索引
- (3) 常查询数字建立索引或者组合索引
- (4) 分解大连接查询, 分解成对每一个表进行一次单表查询. 可以对缓存更高效的运用. 即使其中一个表发生变化, 对其他表的查询缓存依然可以使用.

9.1.0.18 InnoDB & MyISAM

(1) myISAM 支持全文索引, innodb 也支持全文索引, 简单来说innodb 比myISAM强大太多了

另外innodb 对于全文索引有要求, 有一个最小搜索长度. 和最大搜索长度. 比like 之类快很多.

9.1.0.19 创建存储过程

- (1) 创建存储过程比单独的sql语句要快
- (2) 可以快速进行测试

9.1.0.20 热备份和冷备份

冷备份: 因为mysql是基于文件的, 所以, 我们在mysql关闭的时候, 直接使用 copy 拷贝一份即可.

热备份: 使用mysqldump 命令对正在运行的mysql程序的数据库进行备份

9.1.0.21 Innodb加锁

(1) 对于update,delete和insert语句, Innodb会自动给设计数据集加排它锁(X), 对于普通select语句Innodb不会加任何锁; 事物可以通过以下语句显示给记录集加共享锁或排他锁. (2) 共享锁(S): select * from tableName where ... LOCK IN SHARE MODE(可以查看但无法修改和删除一种数据锁, 其他用户可以加共享锁)

(3) 排它锁(X): SELECT * from tableName where ... FOR UPDATE(独占锁, 其他任何事物都不能对A加任何类型的锁, 直到自己释放了锁.)

共享锁, 主要用在确保没人对这个记录进行UPDATE或者DELETE操作.

9.1.0.22 INNODB解决死锁

实际上在INNODB发现死锁之后,会计算出两个事物各自插入,更新或者删除的数据量来判定两个事物的大小.也就是哪个事物所改变的记录条数越多,在死锁中就越不会被回滚掉.

9.1.0.23 Mysql锁你了解哪些

按照锁的粒度区分

1. 行锁, 锁某行数据, 锁力度最小, 并发度最高
2. 表锁, 锁整张表, 锁力度最大, 并发度低
3. 间隙锁, 锁的是一个区间

按照怕他性区分

1. 共享锁: 也就是读锁
2. 排它锁: 也就是写锁

还可以分为:

1. 乐观锁, 并不会真正的去锁某行记录, 而是通过一个版本号来实现的.
2. 悲观锁, 上面所说的行锁, 表锁, 都是悲观锁.

9.1.0.24 Mysql数据库中,什么情况下设置了索引但是无法使用?

1. 没有符合最左前缀原则
2. 字段进行了隐私数据类型转化
3. 走索引没有全表扫描效率高

10

操作系统

► ...

10.1 知识点和方法论

10.1.0.1 共享内存

共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

10.1.0.2 虚拟内存

对虚拟内存的定义是基于对地址空间的重定义的，即把地址空间定义为「连续的

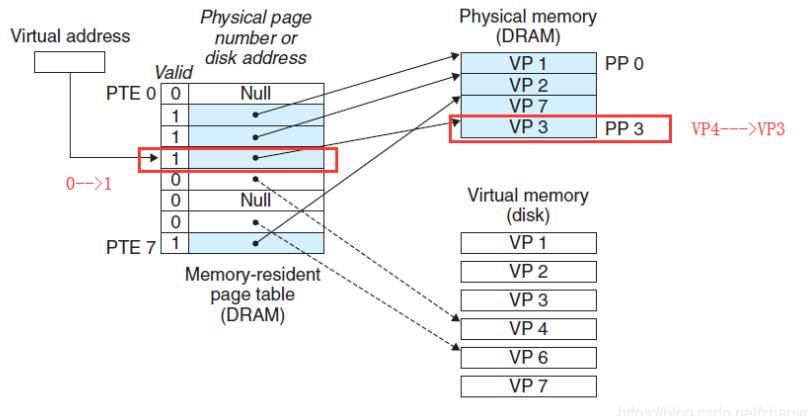


图 10-1 virtualMemory

虚拟内存地址」，以借此「欺骗」程序，使它们以为自己正在使用一大块的「连续」地址。

也就是说虚拟内存能提供一大块连续的地址空间，对程序来说它是连续的，完整的，实际上虚拟内存是映射在多个物理内存碎片上，还有部分映射到了外部磁盘存储器上。虚拟内存有以下两个优点：

1. 虚拟内存地址空间是连续的，没有碎片
2. 虚拟内存的最大空间就是cpu的最大寻址空间，不受内存大小的限制，能提供比内存更大的地址空间

当每个进程创建的时候，内核会为每个进程分配虚拟内存，这个时候数据和代码还在磁盘上，当运行到对应的程序时，进程去寻找页表，如果发现页表中地址没有存放在物理内存上，而是在磁盘上，于是发生缺页异常，于是将磁盘上的数据拷贝到物理内存中并更新页表，下次再访问该虚拟地址时就能命中了。

找到进程对应的页表中的条目

10.1.0.3 上下文信息

进程上下文包括三个，用户级上下文，寄存器上下文和系统级上下文

用户级上下文：指令，数据，共享内存、用户栈

寄存器上下文：程序计数器，通用寄存器，控制寄存器，状态字寄存器，栈指针（用来指向用户栈或者内存栈）

系统级上下文：pcb(进程控制块)，主存管理信息（页表&段表）、核心栈

PCB：

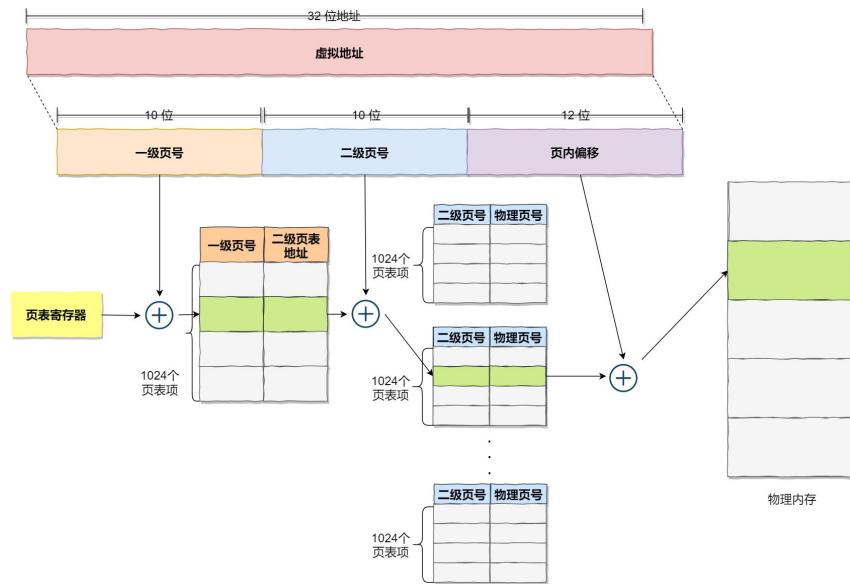


图 10-2 mmu

表示进程的状态. 进程的优先级. pid 进程号

10.1.0.4 内存访问

TLB: 快表, 防止多级转换

MMU: 内存管理单元将虚拟地址转为真实地址

虚拟地址: 页号+页内偏移; 可能产生多级页表: 一级页号+二级页号+业内偏移

物理地址: 硬盘上的地址

首先CPU先使用虚拟地址通过MMU转为真实地址, 查找TLB快表上是否有存储真实地址的值, 如果存在, 那么直接去拿到真实地址的值, 如果不存在那么: 虚拟地址第一级页号找到二级页表地址. 然后通过二级页好找到二级页表内的地址, 二级页表内存储了真实的物理页号, 找到物理页号+虚拟地址的偏移找到真实地址

10.1.0.5 匿名管道和命名管道的区别

匿名管道由pipe函数创建并打开, 只能由父子进程进行半双工传递

命名管道由mkfifo函数创建, 打开用open

10.1.0.6 线程和进程的区别

1. 一个线程从属于一个进程; 一个进程可以包含多个线程

2. 进程是系统资源调度的最小单位; 线程CPU调度的最小单位
3. 进程系统开销显著大于线程系统开销; 线程所需要的系统资源更少
4. 进程在执行时候拥有独立的内存空间, 多个线程共享进程的内存, 如代码段, 数据段, 扩展段; 但每个线程拥有自己的栈段和寄存器组
5. 进程切换开销
 - a. 切换页表全局目录(比较费时, 对于大的进程之间的切换.)
 - b. 切换内核态堆栈
 - c. 切换硬件上下文
6. 线程切换开销

因为线程是进程的子集, 如果同一个进程之间的线程切换, 上下文会比进程切换要小.

10.1.0.7 协程与线程进行比较

协程, 是一种用户态的轻量级线程, 写成的调用完全由用户控制. 协程调度切换时, 将寄存器上下文和栈保存到其他地方, 在切回来的时候, 恢复先前保存的寄存器上下文和栈, 直接操作栈则基本没有内核切换开销, 可以不加锁访问全局变量.

- (1) 一个线程可以有多个协程, 一个进程也可以单独拥有多个协程.
- (2) 线程进程都是同步机制, 而协程则是异步

10.1.0.8 进程之间的通信方式有哪些?

- (1) 命名管道FIFO: 半双工方式
- (2) 消息队列: 克服了信号传递信息少, 管道只能承载无格式字节流以及缓冲区大小受限等缺点.
- (3) 共享内存: 共享内存是最快的IPC方式.
- (4) 信号量: 信号量是一个计数器.
- (5) 套接字:socket 可以用于不同机器间的进程通信.
- (6) 信号: 信号是一种比较复杂的通信方式, 用于通知接收进程某个事件已经发生, 比如linux中的kill命令通知进程进行关闭.

10.1.0.9 进程调度算法

- (1) 先来先服务FCFS
- (2) 短作业有限SJF

- (3) 高优先权优先, 非抢占式优先权算法 & 强占式优先权调度算法
- (4) 高响应比优先调度算法可以克服SJF长作业的饥饿
- (5) 基于时间片轮转调度算法

10.1.0.10 epoll和poll的区别

1. select模型, 使用的是数组来存储Socket连接文件描述符, 容量是固定的, 需要通过轮询来判断是否发生了IO事件
2. poll模型, 使用的是链表来存储Socket连接文件描述符, 容量是不固定的, 同样需要通过轮询来判断是否发生了IO事件
3. epoll模型, epoll和poll是完全不同的, epoll是一种事件通知模型, 大发生了IO事件时, 应用程序才进行IO操作, 不需要像poll模型那样主动去轮询

select监控的句柄列表在用户态, 每次调用都需要从用户态将句柄列表拷贝到内核态, 但是epoll中句柄就是建立在内核中的, 这样就减少了内核和用户态的拷贝, 高效的原因之一。

还会再建立一个list链表, 用于存储准备就绪的事件.

当epoll_wait调用时, 仅仅观察这个list链表里有没有数据即可。有数据就返回, 没有数据就sleep, 等到timeout时间到后即使链表没数据也返回。所以, epoll_wait非常高效。

11

设计模式

► ...

11.1 知识点和方法论

11.1.0.1 多线程下单例设计模式

- (1) 饿汉式不会出现安全问题, 懒汉式会出现(同时创建的时候会有问题, 两个线程).
- (2) 懒汉式安全隐患解决
- (3) 饿汉式, 在静态属性中就获取了对象, 懒汉式是去得到对象的时候获取, 导致了懒汉式可能有问题.

```
1 package com.lf.shejimoshi;
2
3 /**
4 * @classDesc: 类描述懒汉式单例测试类:()
5 * @author baobaolan
6 * @createTime 年月日2018110
7 * @version v1.0
8 */
9 public class SingletonTest {
10     /**
11      * @functionDesc: 功能描述测试懒汉式单例模式:()
12      * @author baobaolan
13      * @createTime 年月日2018110
14      * @version v1.0
```

```
15     */
16     public static void main(String[] args) {
17         Student s1 = Student.getStudent();
18         Student s2 = Student.getStudent();
19         System.out.println(s1==s2);
20     }
21
22 }
23
24 /**
25 * @classDesc: 类描述学生类:()
26 * @author baobaolan
27 * @createTime 年月日2018110
28 * @version v1.0
29 */
30 class Student{
31
32     定义全局变量//
33     private static Student student;
34
35     私有化构造函数//
36     private Student(){
37
38     }
39
40 /**
41 * @functionDesc: 功能描述对外暴露方法:()
42 * @author baobaolan
43 * @createTime 年月日2018110
44 * @version v1.0
45 */
46     public static Student getStudent(){
47         if(student==null){
48             加上同步锁, 保证线程安全//
49             synchronized(Student.class){
50                 student = new Student();
51             }
52         }
53         return student;
54     }
55 }
```

```
1 package com.lf.shejimoshi;
```

```
2
3 /**
4 * @classDesc: 类描述测试类:()
5 * @author baobaolan
6 * @createTime 年月日2018110
7 * @version v1.0
8 */
9 public class Singleton2Test {
10
11     public static void main(String[] args) {
12
13         Teacher teacher1 = Teacher.getTeacher();
14         Teacher teacher2 = Teacher.getTeacher();
15         System.out.println(teacher1==teacher2);
16
17     }
18
19 }
20
21 /**
22 * @classDesc: 类描述饿汉式单例:()
23 * @author baobaolan
24 * @createTime 年月日2018110
25 * @version v1.0
26 */
27 class Teacher{
28     //类加载的时候初始化一次//
29     private static final Teacher teacher = new Teacher();
30     //私有化构造函数//
31     private Teacher(){
32         super();
33     }
34     /**
35      * @functionDesc: 功能描述对外暴露的方法:()
36      * @author baobaolan
37      * @createTime 年月日2018110
38      * @version v1.0
39      */
40     public static Teacher getTeacher(){
41         return teacher;
42     }
43
44 }
```

11.1.0.2 为什么在wait代码块中要用while而不用if

因为单个生产者单个消费者，没什么问题。如果是一个生产者两个消费者的话会有问题。

因为线程唤醒的话，会直接在wait()下面执行，然后如果是while的话，可以进入重新判断。否则可能造成数据溢出。

```
1  /*生产和消费
2
3 */
4 package multiThread;
5
6 class SynStack
7 {
8     private char[] data = new char[6];
9     private int cnt = 0; 表示数组有效元素的个数//
10
11    public synchronized void push(char ch)
12    {
13        if (cnt >= data.length)
14        {
15            try
16            {
17                System.out.println生产线
18                程(""+Thread.currentThread().getName()+"准备休眠+");
19                this.wait();
20                System.out.println生产线
21                程(""+Thread.currentThread().getName()+"休眠结束了+");
22            }
23            catch (Exception e)
24            {
25                e.printStackTrace();
26            }
27        }
28        this.notify();
29        data[cnt] = ch;
30        ++cnt;
31        System.out.printf生产线程(""+Thread.currentThread().getName()+"正
32        在生产第+"%d个产品，该产品是d: %c\n", cnt, ch);
33    }
34
35    public synchronized char pop()
36    {
37        char ch;
```

```
35     if (cnt <= 0)
36     {
37         try
38         {
39             System.out.println消费线
程(""+Thread.currentThread().getName()准备休眠+"");
40             this.wait();
41             System.out.println消费线
程(""+Thread.currentThread().getName()休眠结束了+"");
42         }
43         catch (Exception e)
44         {
45             e.printStackTrace();
46         }
47     }
48     this.notify();
49     ch = data[cnt-1];
50     System.out.printf消费线程(""+Thread.currentThread().getName()正在消费第+%个产品，该产品是d: %c\n", cnt, ch);
51     --cnt;
52     return ch;
53 }
54 }

55
56 class Producer implements Runnable
57 {
58     private SynStack ss = null;
59     public Producer(SynStack ss)
60     {
61         this.ss = ss;
62     }
63
64     public void run()
65     {
66         char ch;
67         for (int i=0; i<10; ++i)
68         {
69
70             ch = (char)('a'+i);
71             ss.push(ch);
72         }
73     }
74 }
75
76 class Consumer implements Runnable
```

```
77  {
78      private SynStack ss = null;
79
80      public Consumer(SynStack ss)
81      {
82          this.ss = ss;
83      }
84
85      public void run()
86      {
87          for (int i=0; i<10; ++i)
88          {
89              /*try{
90                  Thread.sleep(100);
91              }
92              catch (Exception e){
93                  }*/
94
95              //System.out.printf("%c\n", ss.pop());
96              ss.pop();
97          }
98      }
99  }
100
101
102 public class TestPC2
103 {
104     public static void main(String[] args)
105     {
106         SynStack ss = new SynStack();
107         Producer p = new Producer(ss);
108         Consumer c = new Consumer(ss);
109
110
111         Thread t1 = new Thread(p);
112         t1.setName("1");
113         t1.start();
114         /*Thread t2 = new Thread(p);
115         t2.setName("2");
116         t2.start();*/
117
118         Thread t6 = new Thread(c);
119         t6.setName("6");
120         t6.start();
```

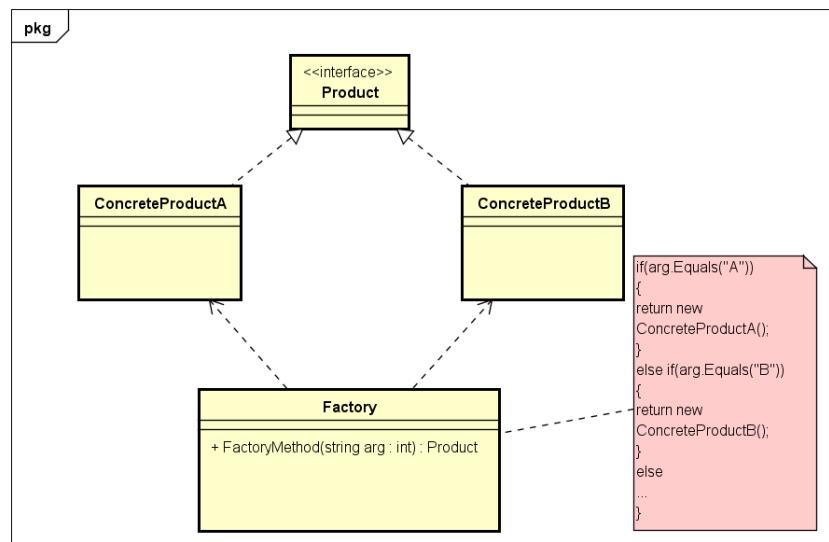


图 11-1 simpleFactory

```

121     Thread t7 = new Thread(c);
122     t7.setName号("7");
123     t7.start();
124 }
125 }
```

11.1.0.3 Serializable

序列化接口, 只有实现这个接口才能序列化, 默认计算一个serialVersionUID, 可以进行自定义.

11.1.0.4 工厂模式

简单工厂模式

简单来说就是使用字符串返回对象

抽象工厂模式

简单来说, 就是工厂也是抽象化的, 原先只有产品是抽象化的

抽象工厂模式的优点分离接口和实现客户端使用抽象工厂来创建需要的对象, 而客户端根本就不知道具体的实现是谁, 客户端只是面向产品的接口编程而已。也就是说, 客户端从具体的产品实现中解耦。

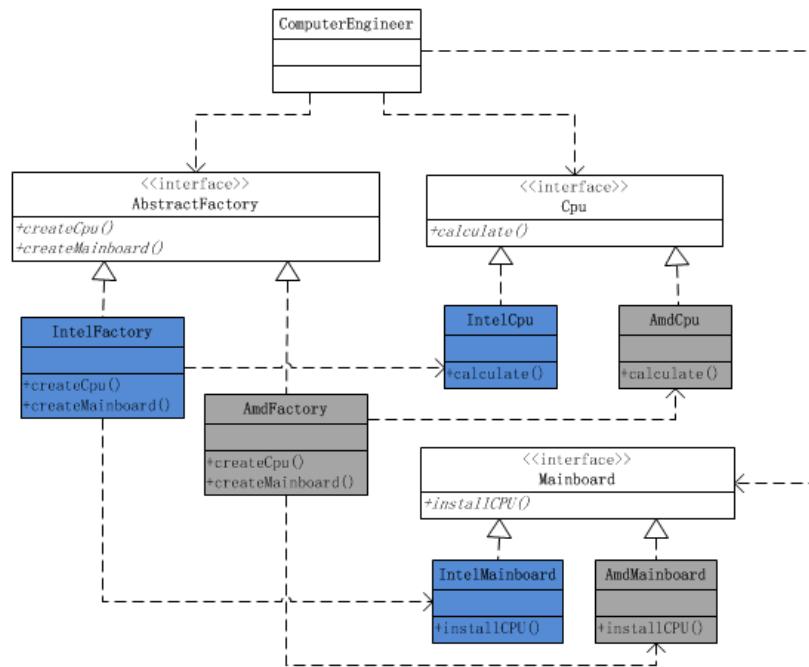


图 11-2 abstractFactory

使切换产品族变得容易因为一个具体的工厂实现代表的是一个产品族，比如上面例子的从Intel系列到AMD系列只需要切换一下具体工厂。

抽象工厂模式的缺点不太容易扩展新的产品如果需要给整个产品族添加一个新的产品，那么就需要修改抽象工厂，这样就会导致修改所有的工厂实现类。

11.1.0.5 单例模式

特点

1、单例类只能有一个实例。2、单例类必须自己创建自己的唯一实例。3、单例类必须给所有其他对象提供这一实例。

饿汉式(线程安全，调用效率高，但是不能延时加载):

```

1 public class ImageLoader{
2     private static ImageLoader instance = new ImageLoader();
3     private ImageLoader(){}
4     public static ImageLoader getInstance(){
5         return instance;
6     }
7 }
```

DCL也就是双重锁判断机制（由于JVM底层模型原因，偶尔会出问题，不建议使用）

```
1 public class SingletonDemo5 {  
2     private volatile static SingletonDemo5 SingletonDemo5;  
3  
4     private SingletonDemo5() {  
5         }  
6  
7     public static SingletonDemo5 newInstance() {  
8         if (SingletonDemo5 == null) {  
9             synchronized (SingletonDemo5.class) {  
10                 if (SingletonDemo5 == null) {  
11                     SingletonDemo5 = new SingletonDemo5();  
12                 }  
13             }  
14         }  
15         return SingletonDemo5;  
16     }  
17 }
```

11.1.0.6 中介模式

11.1.0.7 代理模式

分为静态代理和动态代理

现在可以看到，代理模式可以在不修改被代理对象的基础上，通过扩展代理类，进行一些功能的附加与增强。值得注意的是，代理类和被代理类应该共同实现一个接口，或者是共同继承某个类。

上面介绍的是静态代理的内容，为什么叫做静态呢？因为它的类型是事先预定好的，比如上面代码中的 Cinema 这个类。下面要介绍的内容就是动态代理。

```
1 abstract class Move{  
2     public abstract void play();  
3 }  
4  
5 class RealMove extends Move{  
6     public void play(){  
7         System.out.println("您正在观看电影(\"《肖申克的救赎》\"");  
8     }  
9 }
```

```
10
11 class Cinema extends Move {
12     RealMove movie;
13     public Cinema(RealMove movie) {
14         super();
15         this.movie = movie;
16     }
17
18
19     @Override
20     public void play() {
21         guanggao(true);
22         movie.play();
23         guanggao(false);
24     }
25
26     public void guanggao(boolean isStart){
27         if ( isStart ) {
28             System.out.println电影马上开始了，爆米花、可乐、口香糖折，快来买啊！
29             ("9.8");
30         } else {
31             System.out.println电影马上结束了，爆米花、可乐、口香糖折，买回家吃
32             吧！ ("9.8");
33         }
34     }
35
36 class ProxyTest {
37     public static void main(String[] args) {
38         RealMove realmovie = new RealMove();
39         Move movie = new Cinema(realmovie);
40         movie.play();
41     }
42 }
```

```
1 interface SellWine {
2     void mainJiu();
3 }
4
5 class MaotaiJiu implements SellWine{
6     public void mainJiu(){
7         System.out.println我卖得是茅台酒("") ;
8     }
}
```

```
9  }
10
11 /*
12 * public static Object newProxyInstance(ClassLoader loader,
13 Class<?>[] interfaces,
14 InvocationHandler h)
15 loader 自然是类加载器
16 interfaces 代码要用来代理的接口
17 h 一个 InvocationHandler 对象
18 InvocationHandler 是一个接口，官方文档解释说，每个代理的实例都有一个与之关联的
19 InvocationHandler 实现类，如果代理的方法被调用，那么代理便会通知和转发给内部的
20 InvocationHandler 实现类，由它决定处理。
21 InvocationHandler 内部只是一个 invoke() 方法，正是这个方法决定了怎么样处理代理
22 传递过来的方法调用。
23 proxy 代理对象
24 method 代理对象调用的方法
25 args 调用的方法中的参数
26 */
27 class GuitaiA implements InvocationHandler {
28     private Object pingpai;
29
30     public GuitaiA(Object pingpai){
31         this.pingpai = pingpai;
32     }
33
34     public Object invoke(Object proxy, Method method, Object[] args)
35         throws Throwable{
36         System.out.println销售开始(" 柜台是:
37             "+this.getClass().getSimpleName());
38         method.invoke(pingpai, args);
39         System.out.println销售结束("");
40         return null;
41     }
42 }
43
44 class TestProxy{
45     public static void main(String[] args) {
46         MaotaiJiu maotaiJiu = new MaotaiJiu();
47         InvocationHandler jinxiao1 = new GuitaiA(maotaiJiu);
48         SellWine dynamicProxy = (SellWine) Proxy.newProxyInstance(
49             MaotaiJiu.class.getClassLoader(),
50             MaotaiJiu.class.getInterfaces(), jinxiao1);
51
52         dynamicProxy.mainJiu();
```

```
50 }  
51 }
```

代理分为静态代理和动态代理两种。

静态代理，代理类需要自己编写代码写成。

动态代理，代理类通过 `Proxy.newInstance()` 方法生成。

不管是静态代理还是动态代理，代理与被代理者都要实现两样接口，它们的实质是面向接口编程。

静态代理和动态代理的区别是在于要不要开发者自己定义 `Proxy` 类。

动态代理通过 `Proxy` 动态生成 proxy class，但是它也指定了一个 `InvocationHandler` 的实现类。

代理模式本质上的目的是为了增强现有代码的功能。

Java动态代理只能代理接口，要代理类需要使用第三方的CLIGB等类库。

(动态代理优点) 传统面向对象思想中，如果想要实现功能复用，要么继承、要么引用，无论哪种方式，对代码都有一定的侵入性，耦合无可避免，侵入性啥意思？简单来说：如果你想要用它增强你程序的功能，你必须改动你的程序代码，那它就具有侵入性。如果只有一点两点需要增强还好说，如果大量的功能点需要被增强，工作量就会很大，代码也不太优雅。想象一下，如果你对外公开了一系列的接口，现在领导说了，接口要加权限控制。在哪加？最笨的当然就是写个程序验证的逻辑，然后每个接口都拿来调用一遍。这也正是面向对象思想的短板，在要为程序新增一些通用功能时，只能通过耦合的方式才能进行。AOP正是为此而生，AOP旨在通过一种无耦合的方式来为程序带来增强。而动态代理，就是AOP实现方式中的一种，第2点决定了代理类需要把被代理类耦合进来，这意味着什么呢，代理类跟一个固定的被代理类绑定死了，如果你要对一百个类进行代理，即使代理逻辑一样，你也得编写一百个代理类(静态代理).

至此，Cglib动态代理的原理我们就基本搞清楚了，代码细节有兴趣可以再研究下。最后我们总结一下JDK动态代理和Glib动态代理的区别：

1.JDK动态代理是实现了被代理对象的接口，Cglib是继承了被代理对象。

2.JDK和Cglib都是在运行期生成字节码，JDK是直接写Class字节码，Cglib使用ASM框架写Class字节码，Cglib代理实现更复杂，生成代理类比JDK效率低。

3.JDK调用代理方法，是通过反射机制调用，Cglib是通过FastClass机制直接调用方法，Cglib执行效率更高。

12

C++

► ...

12.1 知识点和方法论

12.1.0.1 map 和 unordered_map 不同点

map 内部实现了一颗红黑树, 红黑树有自动排序的功能.

unordered_map 内部实现了一个哈希表, 如果遇到冲突, 同位置上的元素节点小于8个的时候, 这些数据会是一个链表式的方式互相连起来。当同位置上的元素节点大于8个的时候, 会自动转化为成一个map(红黑树)。

12.1.0.2 C++ 不能重载的运算符

. :: .*? : *sizeof*

大多数运算符都是可以重载的,但是有5个运算符C++语言规定是不可以重载的.

1. .(点运算符),通常用于去对象的成员,但是->(箭头运算符),是可以重载的
- 2.:::(域运算符),即类名+域运算符,取成员,不可以重载
- 3..*(点星运算符,)不可以重载,成员指针运算符".*",即成员是指针类型
- 4.?:(条件运算符),不可以重载
- 5.sizeof,不可以重载

12.1.0.3 C++ 如果一个类中没有任何属性, 占用空间为1, 这是规定

不是规定哦, 为了确保两个不同对象的地址不同, 必须如此, 类的实例化是在内存中分配一块地址, 同样空类也会目次, 所以编译器会给空类隐含的添加一个字节, 这样空类是细化后就有独一无二的地址了. 所以, 空类的sizeof为1, 而不是0.

12.1.0.4 返回引用

优点:

1. 防止返回对象的时候调用拷贝构造函数和析构函数导致不必要的开销, 降低赋值运算符等的效率
2. 允许进行连续赋值

12.1.0.5 类和结构体的区别

默认继承访问权限区别, 类默认是private, struct 默认是 public.

简单来说, 抽象思维来判定是否要使用struct 或者是class, struct 是数据结构, class 是对象.

class 可以用来定义模板参数, struct 不可以

12.1.0.6 类的大小

```

1 class A{};
2 class B{
3     virtual void f(){}; 
4 };
5 class C: public B{};
6 class D: public virtual A{ // 虚继承 防止 菱形继承造成奇异
7 };
8
9 int main() {
10     cout << sizeof(A) << endl;//1
11     cout << sizeof(B) << endl;//8
12     cout << sizeof(C) << endl;//8 虚函数表的地址 因为是位系统所
13         以64是sizeof8
14     cout << sizeof(D) << endl;//8 因为包含指向虚基类的指针
}

```

12.1.0.7 C++ 内存对齐

简单来说就是，默认是4字节对齐。对于一个结构体而言，里面的顺序会对对齐字节产生影响。参考链接 <https://zhuanlan.zhihu.com/p/30007037>

12.1.0.8 C++11

auto 类型推到

auto a = 10;

众所周知C++11新增了右值引用，这里涉及到很多概念：

左值：可以取地址并且有名字的东西就是左值。

右值：不能取地址的没有名字的东西就是右值。

纯右值：运算表达式产生的临时变量、不和对象关联的原始字面量、非引用返回的临时变量、lambda表达式等都是纯右值。

返回值优化：当函数需要返回一个对象实例时候，就会创建一个临时对象并通过复制构造函数将目标对象复制到临时对象，这里有复制构造函数和析构函数会被多余的调用到，有代价，而通过返回值优化，C++标准允许省略调用这些复制构造函数。

std::function & std::bind & lambda表达式

std::thread 线程相关智能指针

委托构造函数

继承构造函数 using Base::Base

nullptr

final & override

default and delete

explicit 只能显示构造不能隐式转换。

修饰类成员函数，表示在该函数内不可以修改该类的成员变量。**const** 比如 void func() const;

constexpr 运行期间就会被运算出来就是常量，否则是普通函数。

minmax_element: 返回容器内最大元素和最小元素位置

<https://zhuanlan.zhihu.com/p/139515439>

12.1.0.9 C++14特性

auto 返回值推导

支持变量模板

[[deprecated]] 警告标记

12.1.0.10 C++17特性

C++17正式将file_system纳入标准中，提供了关于文件的大多数功能，基本上应有尽有

12.1.0.11 C++20特性

```
auto res = foo <=> bar;  
import 和 export
```

12.1.0.12 模板类的函数定义

```
1 template <typename T>  
2 void vector<T>::clear() {  
3  
4 }
```

12.1.0.13 尾递归的实现

简单来说，如果直接返回结果的话会造成多次调用，但是把返回的结果放在参数中，编译器会进行一定的优化，将其转为非递归的形式。然后我们就可以减少爆栈的风险。

尾递归由于直接返回值，不需要保存临时变量，所以性能不会产生线性增加。并且编译器会将尾递归形式优化成非递归形式。

12.1.0.14 socket 如何实现可靠连接

1. 心跳包机制，保持连接

2. 可以在正式数据发送前发送监测包，刻骨短收到检测包后立即发挥一个响应。服务端收到响应，认为客户端还活着，把正式数据发送

12.1.0.15 set, multiset, map, multimap

底层数据结构使用红黑树，map/multimap使用pair作为基础元素。set/multiset value和key相同

底层红黑树multi/nonulti 使用 insert_unique 和 insert_equal之间的差别

12.1.0.16 定义的静态全局变量作用于是

本文件

12.1.0.17 如何判断一段程序是由C编译器编译还是由C++编译器编译的

有内置宏 `__cplusplus` 是C++编译的

12.1.0.18 在C++程序中调用被C编译器编译后的函数, 为什么要加`extern "C"`

`extern "C"` 是修饰的变量和函数是按照C语言方式编译和链接的, 因为C编译器和C++编译器对一个函数的编译后的函数名是不同的, 这样为了实现混合查找函数名在类库中的实现, 解决名字匹配问题.

12.1.0.19 C++, const规则

`const`只对它左边的东西起作用, 唯一的例外就是`const`本身就是最左边的修饰符, 那么它才会对右边的东西起作用。根据这个规则来判断就很容易了

12.1.0.20 C++, const 和 `#define` 之间的区别

`const`和`#define`都能定义常量, 但是`#define`只做单纯的替换, 但是`const`能进行代码安全检查.

12.1.0.21 指针和引用之间的区别

1. 指针可以指向空值但是引用不能指向空值.
2. 指针可以不初始化, 引用必须初始化.
3. 指针可以随时更改指向的目标, 而引用初始化后就不可以再指向任何其他对象
4. 指针可以进行地址偏移, 引用不可以

12.1.0.22 inline的优劣

简单来说, 减少了函数调用, 但是增大了生成可执行程序的体积

12.1.0.23 C++11有什么你使用到的新特性

`auto` 遍历的时候很方便对于一些很复杂的变量直接使用`auto`, 让编译器去推断他的类型

lambda 表达式, 比如在sort中可以很方便的写出cmp函数

12.1.0.24 C++中有malloc/free, 为什么还需要new/delete

malloc/free是C/C++标准库函数, new/delete是C++运算法。他们都可以用于申请和释放内存。

对于类类型的对象而言, malloc/free 无法满足要求, 不会自动执行构造和析构函数。因为C++需要new/delete

12.1.0.25 面向对象技术的基本概念是什么, 三个基本特征是什么?

基本概念: 类、对象、继承; 基本特征: 封装、继承、多态。

12.1.0.26 为什么基类的析构函数是虚函数?

当我们使用基类的指针管理派生类, 使用delete释放该指针时, 会调用基类的析构函数 Base(), 如果基类的析构函数是虚函数, 那么就会继续调用派生类的析构函数 Derived(); 而如果基类的析构函数不是虚函数, 就只会调用基类的析构函数, 那么派生类中的那片内存就不会被释放, 从而造成内存泄漏。

12.1.0.27 为什么构造函数不能为虚函数?

答: 虚函数采用一种虚调用的方法。虚调用是一种可以在只有部分信息的情况下工作的机制。如果创建一个对象, 则需要知道对象的准确类型, 因此构造函数不能为虚函数。

12.1.0.28 如果虚函数是有效的, 那为什么不把所有函数设为虚函数?

答: 不行。首先, 虚函数是有代价的, 由于每个虚函数的对象都要维护一个虚函数表, 因此在使用虚函数的时候都会产生一定的系统开销, 这是没有必要的。

12.1.0.29 什么是多态? 多态有什么作用?

答: 多态就是将基类类型的指针或者引用指向派生类型的对象。多态通过虚函数机制实现。多态的作用是接口重用。

12.1.0.30 重载和覆盖有什么区别？

答：虚函数是基类希望派生类重新定义的函数，派生类重新定义基类虚函数的做法叫做覆盖；

重载就在允许在相同作用域中存在多个同名的函数，这些函数的参数表不同。重载的概念不属于面向对象编程，编译器根据函数不同的形参表对同名函数的名称做修饰，然后这些同名函数就成了不同的函数。重载的确定是在编译时确定，是静态的；虚函数则是在运行时动态确定。

12.1.0.31 什么是虚指针？

答：虚指针或虚函数指针是虚函数的实现细节。带有虚函数的每一个对象都有一个虚指针指向该类的虚函数表。

12.1.0.32 main函数执行之前会执行什么？执行之后还能执行代码吗？

1. 全局对象的构造函数会在main函数之前执行
2. 可以，可以用atexit注册一个函数(函数参数是一个函数指针)，它会在main之后执行；

12.1.0.33 经常要操作的内存分为那几个类别？

(1) 栈区：由编译器自动分配和释放，存放函数的参数值、局部变量的值等；向下增长它的生长方式是向下的，是向着内存地址减小的方向增长。

(2) 堆：一般由程序员分配和释放，存放动态分配的变量；向上增长也就是向着内存地址增加的方向；

(3) 全局区(静态区)：全局变量和(全局或局部)静态变量存放在这一块，初始化的和未初始化的分开放(.data 存放初始化后的数据 .bss 存放未初始化的数据)；

- (4) 常量区：常量字符串就放在这里，程序结束自动释放(.text 分区)；
- (5) 程序代码区：参访函数体的二进制代码。

//=====另一本书编写

1. 栈：在函数调用时存储参数，返回地址和局部变量
2. 内存映射区：文件到内存的映射和程序运行时申请的较大数据
3. 堆：存储程序运行时申请的较小数据，动态分配和销毁
4. BSS段：存储未被初始化的全局变量和静态变量，长度固定
5. 数据段：存储已被初始化的全局变量和静态变量，长度固定

6. 代码段: 存储程序的二进制代码, 长度固定

12.1.0.34 链接器

linux 中 g++ -o 将多个.o文件链接成一个可执行文件.

简单的来说, 我们针对一个.cc文件进行编译, 当做函数申明是存在的, 链接就是将这些函数找到. 链接也分为动态库链接.so, 和静态库链接 .a. 静态库链接直接打包的可执行文件中. 动态链接是, 不打包到文件中. 我们每次要用到动态库里面函数. 操作系统会帮我们找到动态库在哪里.

12.1.0.35 符号表的作用

编译器会生成一个叫做“符号表”的数据结构来维护变量名和内存地址直接的对应关系。它会搜集变量名, 比如我们定义了一个全局的 int a; 那么编译器会为程序预留4个字节(32位平台)的空间, 比如起始地址23456788(长度为4), 并把变量名“a”和地址88888888保存进符号表, 这样程序中对a进行相关操作时, 它就会根据符号表找到变量的真正的物理位置(23456788), 进行相关操作。在机器执行程序的时候, 会把变量名替换为内存地址(和长度), 而不存在任何名称。

12.1.0.36 函数指针与指针函数

指针函数: 是函数, 但是返回指针(有两个括号)

函数指针: 是指针, 指向函数, 有四个括号

12.1.0.37 内部连接和外部链接有什么区别?

1. 如果变量是内部链接的话, 那么此变量只能在当前文件内访问
2. 如果变量是外部链接的话, 那么此变量可以被其他文件使用

—
静态全局变量默认是内部链接, 而extern默认是外部链接

12.1.0.38 声明与定义的区别

声明, 表示告诉编译器这个符号是存在的, 你先让我编译通过, 让连接器去找这个符号在哪里

对于变量来说, 定义就是声明, 对于函数来说是有区别的, 如果没有实现函数体, 那么就是声明, 表示有这么一个函数. 至于函数在哪里.

12.1.0.39 编译链接过程

1. 预编译, 将#include 和 #define 展开, 生成.i文件
2. 编译, 进行词法分析, 语法分析, 语义分析, 中间代码生成, 目标代码生成, 优化, 生成.s文件
3. 汇编, 生成.o文件, 将汇编码翻译成机器码
4. 链接, 地址和空间分配, 生成.out 文件

12.1.0.40 C++函数中值的传递方式有哪几种?

三种传递方式为：值传递、指针传递和引用传递。

12.1.0.41 信号量和互斥量

在C++中互斥量是mutex, 当一个线程获得了锁, 之后其他线程就不能访问到这个资源, 线程阻塞. 直到获得资源的线程unlock

在C++中叫做Semaphore, 信号量可以有多个, 如果值大于0, 则获得, 值减1, 如果值等于0, 则线程进入睡眠状态直到信号量大于0.

锁是服务于共享资源的; 而semaphore是服务于多个线程间多个资源的执行的逻辑顺序的。

12.1.0.42 RVO和NRVO

RVO(return value optimization) 返回值优化, 防止产生临时对象.

```

1 Point3d factory()
2 {
3     Point3d x;
4     return x;
5 }
6 Point3d p = factory();

```

优化成

```

1 Point3d p;
2 factory(p);
3 factory(const Point3d &_result)
4 {
5     Point3d x;
6     result.Point3d::Point3d(x); 复制构造函数// 还没有吧这个名字优化掉x

```

```

7     return;
8 }
```

NRVO(name return value optimization) NRVO的优化比RVO 的优化更进一步，直接将要初始化的对象替代掉返回的局部对象进行操作。可以看出，进行NRVO 的优化后，此时整个函数将会只调用一次构造函数。

```

1 Point3d p;
2 factory(p);
3 factory(const Point3d &_result)
4 {
5     result.Point3d::Point3d(x); // 直接操作参数，没有了x
6     return;
7 }
```

12.1.0.43 听说过mangling么？

简单来说，就是C++为了函数重载实现的名称修改，有一定的规则，比如_Z4funii

12.1.0.44 模板代码如何组织？模板的编译以及实例化过程？

模板类的声明要全部放在头文件中。

12.1.0.45 C++中四种Cast的使用场景

`static_cast<xxx>()`：表示编译级别的强制类型转换，且不能发现运行时的错误。类似C的(int)之类的强制转圈，不能去除const属性，volatile 属性。还有一个unaligned属性

`dynamic_cast<>()`：运行时检查类型。主要用于含有虚函数的父类和子类之间的指针转换。会检查是否能够完成这次转换，如果不能返回0

`const_cast<>()`：作为`static_cast`的补充，可以去除const属性

`reinterpret_cast<>()`：低层次的类型转换，可以将指针转为int类型或者long类型。

12.1.0.46 C++什么是常量折叠

简单来说，我定义了一个`const int` 变量，然后我对这个变量进行了修改，(用类型转化啥的) 然后，输出值的时候还是原来的值，因为输出的时候，直接替换为了常量值。其实值是被修改了的。

12.1.0.47 为什么const修饰成员函数后不能修改成员变量

每个成员函数在调用的时候，都会把this作为第一个参数传进去。我们在用const修饰成员函数的时候，就相当于修饰了this，也就是说我们的第一个参数应该是 const 类型 * this；

12.1.0.48 auto_ptr 被弃用了

因为可能导致对同一块堆空间多次delete

12.1.0.49 const int*

```
const int * a1 = &b; // 相当于 *a1 是固定的, a1是可变的  
int *const a2 = &b; // 相当于 a2 是固定的, *a2是可变的  
int const *a3 = &b; // 等价于第一个 a1
```

12.1.0.50 C++智能指针

为什么要使用智能指针？

因为当传递过来一个指针对象，我们不知道，是否要进行资源的释放。std::unique_ptr - 只有一个能拥有这个资源

Std::shared_ptr - 很多个人能拥有这个资源但是这个资源的最后持有者会进行资源的释放操作。

Std::weak_ptr - 简单来说是为了解决shared_ptr循环引用造成资源泄露的问题。

12.1.0.51 C++与java的不同点

C++基于RTTI的机制进行运行时类型识别，简单的说就是typeid + 四个cast进行类型的判断。简单来说，typeid如果对于静态变量是静态编译的时候就可以做类型判定，但是对于动态变量，要运行时判定，会造成一定的影响。

```
1 class A{  
2     public:  
3         virtual void print() {  
4             cout << "I'm A\n";  
5         }  
6     };  
7 class B : public A {  
8     public:
```

```

9  void print() {
10    cout << "I'm B\n";
11  }
12};

13
14 class C : public A {
15   public:
16     void print() {
17       cout << "I'm C\n";
18     }
19};

20
21 int main () {
22   A *p = new B();
23   p->print();
24   cout << typeid(p).name() << endl;
25   if(typeid(p) == typeid(B*)) {
26     cout << "p point b\n";
27   }else if(typeid(p) == typeid(C*)) {
28     cout << "p point c\n";
29   }else if(typeid(p) == typeid(A*)) {
30     cout << "p== \n";
31   }
32   //C * pc = dynamic\_cast<C*>(p);
33   // cat't succ
34   // if(pc){
35     //   pc->print();
36     // }
37   B * pb = dynamic\_cast<B*>(p);
38   if(pb){
39     pb->print();
40   }
41   const int a = 10;
42   cout << typeid(\&a).name() << endl; //const int *
43   cout << typeid(a).name() << endl; //const int *
44   return 0;
45 }
```

java 也有RTTI, 使用 instanceof, 或许加上反射(反射是大部分操作)

```

1 class Base {}
2 class Derived extends Base {}
3
4 public class demo {
```

```
5  public static void main(String[] args) {  
6      Derived derived = new Derived();  
7      System.out.println(derived instanceof Base); // 输出true  
8  }  
9 }
```

```
1 class Customer {  
2     private Long id;  
3     private String name;  
4     private int age;  
5  
6     public Customer() {}  
7  
8     public Customer(String name,int age) {  
9         this.name = name;  
10        this.age = age;  
11    }  
12  
13    public Long getId() {  
14        return id;  
15    }  
16    public void setId(Long id) {  
17        this.id=id;  
18    }  
19    public String getName() {  
20        return name;  
21    }  
22    public void setName(String name) {  
23        this.name=name;  
24    }  
25    public int getAge() {  
26        return age;  
27    }  
28    public void setAge(int age) {  
29        this.age=age;  
30    }  
31  
32 }  
33  
34 class ReflectTester {  
35     public Object copy(Object object) throws Exception{  
36         // 获取对象类型  
37         Class classType=object.getClass();
```

```
38     System.out.println("Class:" + classType.getName());
39
40     // 通过默认构造方法创建一个新的对象
41     Object objectCopy = classType.getConstructor(new Class[]{}).
42         newInstance(new Object[]{});
43
44     // 获取对象的所有属性
45     Field fields[] = classType.getDeclaredFields();
46
47     for(int i=0; i<fields.length; i++){
48         Field field = fields[i];
49         String fieldName = field.getName();
50         String firstLetter = fieldName.substring(0,1).toUpperCase();
51         System.out.println("firstLetter " + firstLetter);
52         获得和属性对应的//getXXX()方法的名字
53         String getMethodName = "get" + firstLetter + fieldName.substring
54             (1);
55         // 获得和属性对应的setXXX()方法的名字
56         String setMethodName = "set" + firstLetter + fieldName.substring
57             (1);
58
59         // 获得和属性对应的getXXX()方法
60         Method getMethod = classType.getMethod(getMethodName, new
61             Class[]{});
62         // 获得和属性对应的setXXX()方法
63         Method setMethod = classType.getMethod(setMethodName, new
64             Class[] {field.getType()});
65         System.out.println("field.getType() " + field.getType());
66         // 调用原对象的getXXX()方法
67         Object value = getMethod.invoke(object, new Object[]{});
68         System.out.println(fieldName + ":" + value);
69         // 调用拷贝对象的setXXX()方法
70         setMethod.invoke(objectCopy, new Object[] {value});
71     }
72     return objectCopy;
73 }
74 }
```

12.1.0.52 C++异常和java异常处理的对比

C++ 没有 finally, 因为c++中有一个非常重要的原则, 就是: 在异常发生前成功调用构造函数的类一定能够执行析构函数。

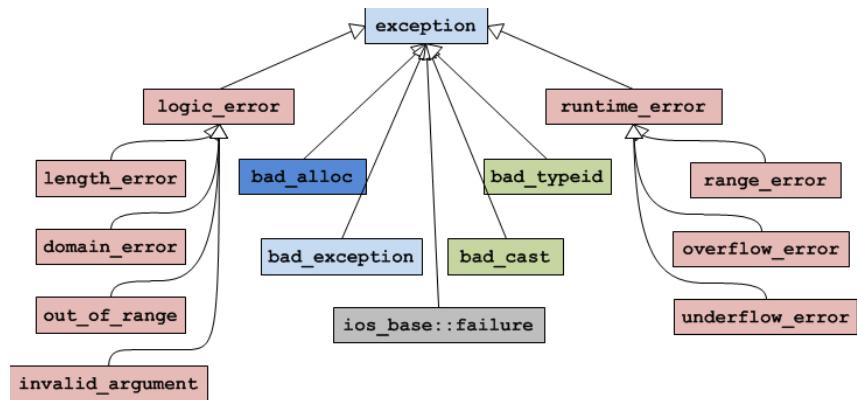


图 12-1 exception_cpp

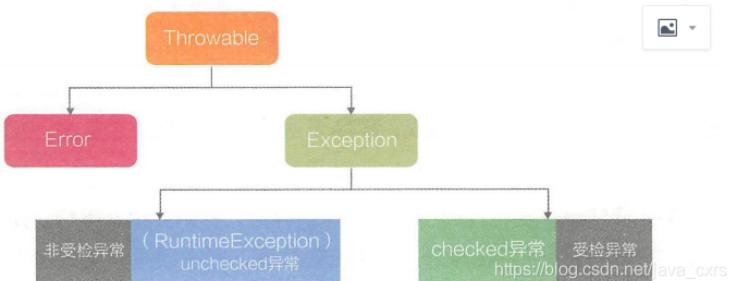


图 12-2 exception_java

但是不能捕获除0异常，需要通过信号来实现 <http://blog.bitfoc.us/p/100>
 java 继承于error 和 exception , exception 的异常能够捕获.

throw用在方法内，用来抛出一个异常对象，将这个异常对象传递到调用者处，并结束当前方法的执行。

使用的格式：

throw new 异常类名(参数);

(throws)运用于方法声明之上，用于表示当前方法不处理异常，而是提醒该方法的调用者来处理异常

使用格式：

修饰符返回值类型方法名(参数) throws 异常类名1, 异常类名2 ...

12.1.0.53 C++虚函数原理

1. 简单来说, 每一个含有虚函数的类都至少有一个与之对应的虚函数表, 其中存放着该类所有的虚函数对应的函数指针
2. 在运行时, 会根据调用的指针指向的对象得到真正应该调用的函数, 然后通过偏移量找到虚函数地址并调用

12.1.0.54 C++虚函数表的开销

1. 空间开销, 每个对象都会保持一个虚函数表造成空间开销
2. 时间开销, 可能因为函数数简介寻址, 造成CPU分支预测失败造成流水线重刷性能开销

12.1.0.55 epoll水平触发&epoll边缘触发

对于监听的sockfd, 最好使用水平触发模式, 边缘触发模式会导致高并发情况下, 有的客户端会连接不上。如果非要使用边缘触发, 网上有的方案是用while来循环accept()。

LT模式

fd可读之后, 如果服务程序读走一部分就结束此次读取, LT模式下该文件描述符仍然可读

fd可写之后, 如果服务程序写了一部分就结束此次写入, LT模式下该文件描述符也仍然可写

ET模式

fd可读之后, 如果服务程序读走一部分就结束此次读取, ET模式下该文件描述符是不可读, 需要等到下次有数据到达时才可变为可读, 所有我们要保证循环读取数据, 以确保把所有数据读出

fd可写之后, 如果服务程序写了一部分就结束此次写入, ET模式下该文件描述符是不可写的, 我们要保证写入数据, 确保把数据写满

12.1.0.56 传统IO和mmap

1. 调用write, 告诉内核需要写入数据的开始地址与长度
2. 内核将数据拷贝到内核页缓存
3. 有操作系统调用, 将数据拷贝到磁盘, 完成写入.

Linux通过将一个虚拟内存区域与一个磁盘上的对象关联起来，以初始化这个虚拟内存区域的内容。

可以减少一次拷贝。

12.1.0.57 write 和 fwrite

如果文件的大小是8k。若用write，且只分配了2k的缓存，则要将此文件读入需要做4次系统调用。（内核空间和用户空间切换4次）若用fwrite，则系统自动分配缓存，则读入此文件只要一次系统调用。也就是用write要读4次磁盘，而用fwrite则只要读1次磁盘。所以fwrite的效率比write要高4倍。

13

项目解析

► ...

13.1 知识点和方法论

13.1.0.1 华为软件精英挑战赛2020

2020年华为软挑是对金融风控的查询, 简单来说实现了一个对于循环转账3-7个账号循环转账的监测. 对于算法时间的优化.

刚开始使用string字符串, 结果因为string字符串基于堆, 生成和释放比较消耗时间, 我们使用了全局静态变量, 用来简化字符串的生成. 以空间换时间. 后来经过测试没有自己写的char数组转int快, 后来换成自己写的转换函数了

使用mmap进行内存映射, 减少IO读取时间(普通的read会拷贝一次到内核缓冲区). 再将整个缓冲区用fwrite写入文件, 因为fwrite带有缓冲区, 比write而言, write使用的是系统缓冲区, 可能会增加IO次数.

使用dfs然后最开始使用了7层dfs+回溯. 十分耗时, 然后, 我实现了6+1层dfs, 减少了一层dfs. 减少一层是通过开始节点的入度判定的, 遍历6层之后, 如果下一个节点是头结点的. 那么结束循环保存答案

后来进一步优化, 使用 5+2(反向) dfs 减少时间. 先反向遍历两层, 然后再正向遍历5层, 当遍历到第5层的时候, 如果是两次反向遍历可以达到head节点, 的话, 我们就输出答案.

进一步优化, 使用反向遍历3进行剪枝, 如果访问超过了3层, 但是反向遍历3层不是头结点, 那么进行剪枝

均匀使用四个区间, 开启四个线程进行运算

13.1.0.2 华为软件精英挑战赛2021

华为软挑是实现云计算背景下的服务器资源分配和调度问题. 核心构成就是策略的使用, 使金额最小

给定一些服务器与虚拟机, 每日会有一定量的创建与删除虚拟机请求, 我们需要合理安排服务器的购买和虚拟机请求的分配, 从而达到购买服务器的成本和能耗成本的总和最低。一台服务器的抽象化为两个CPU分区, 和两个内存分区. 虚拟机也抽象化为可以单部署和双部署, 双部署就是, 对于一台服务器而言要均衡的放在两个CPU分区, 和两个内存分区.

团队是qq群中组建的, 一个浙大的研究生, 两个杭电的研究生. 因为杭电和浙大还是距离挺远的, 我们采用了线上开会的形式. 代码采用了github的方式进行同步. 简单讲一下我们的校验手段, 比赛开始之后一段时间, github上就有了可视化平台, 我们利用可视化平台可以比较详细的看出, 我们购买的服务器的种类, 和购买的个数. 还有整个系统的波动曲线, 就是购买个数之类的. 每日能耗曲线之类的.

这个问题我们探讨出有两个点可以进行集中优化.

A. 即购买策略

- a. 将当天需要购买的虚拟机按照, CPU数量和内存数量的和进行排序, 降序排序
- b. 然后寻找一个服务器开着的, 然后尝试将虚拟机放入服务器, 选择服务器的使用率最高的那个, 进行存放.

- c. 如果没有成功放置进入C环节, 选择一台关机了的服务器, 进行存放.

d. 还没成功选择, 选择一台服务器刚开始我们使用的是, 恰恰能放入虚拟机的服务器. 后期我们进行了更新, 选择, 能放入这台服务器和前面已经存放的服务器的大小, 就是留有一定的余量. 可以减少购买服务器的资金.

B. 迁移的策略

a. 我们每天可以进行一次服务器迁移, 这样将利用率低的服务器从低到高进行排序, 然后将利用率低的服务器迁移到利用率高的服务器.

b. 后期我们增加了一个简单的策略. 将一台服务器中只有一个和两个的挑选出来, 放入其他的服务器中, 这样可以带来省电的效果.

前期我们首先三个人每个人按照自己的想法制作了一个基线版本, 为什么要写一个基线版本呢? 这样我们对整个问题都有一个比较深刻的认识, 后期, 我们更具我们建模

出来的可以比较提升系统性能的地方, 进行了改进. 对于购买策略, 和迁移策略进行改进. 我个人对于购买策略和系统参数进行了调试. 购买策略的排序和余量设定, 给我的系统带来了大约2000w的资金节省. 调试参数, 就是在题目要求的时间边缘进行反复测试直到达到一个最优值, 前期我们设定的参数是低于30% 的利用率我们会进行迁移, 这样迁移的机器数量比较少, 并不能达到我们的预期, 后期我进行慢慢测试终于选定了50%这个参数, 选了更大的参数大部分服务器都要进行迁移, 反而效果并不是特别好.

迁移策略由另外两个同学进行改进. 迁移策略我印象比较深刻的地方是, 因为排行榜上前几名的系统迁移基本上拉满了, 我们还有很多的余量. 所以他们连个进行了这个开发. 他们使用了类似于快速排序点的方法进行迁移. 按照服务器日常能耗的价格进行排序, 然后设定两个指针, 一个指针指向价格比较低的地方, 一个指针指向价格高的地方, 然后尝试将价格高的服务器迁移到价格低的服务器上, 如果成功, 然后右指针`-`, 如果失败左指针`++`. 直到两个指针相遇. 但是因为比较容易超时, 我们放弃了这个方案.

对于迁移, 我们自定义了服务器性价比. 使用服务器的CPU利用率 * 内存利用率, 如果其中有一个特别低的话, 就说明这台服务器没有达到比较大的利用率, 我们设定一个参数, 当CPU利用率 * 内存利用率小于50% 我们会对这个服务器中的虚拟机进行迁移.

13.1.0.3 数学建模2020

1. 数学建模, 问题是对于飞机6个油箱的输油拟合, 我们采用了向量拟合算法, 判断重心之间的偏差, 然后控制邮箱的输油, 使用python有一个好处可以将数据读取和数据处理和数据展示写在一堆中, 队友对于文章的撰写, 也很厉害, 队友使用visio绘制了形象生动的模型展示图. 是一个团队的比赛, 然后我觉得最重要的是心态平和, 在最后一天晚上, 凌晨两点的时候我才完成了数学建模最后一问的撰写. 当时队友都说, 要不先不写最后一题了, 我觉得我写了出来, 而且不超过2点, 结果, 我们就完成了数学建模的所有题目。

2. 还有经过数学建模, 个人觉得, 只要有合适的学习阶梯, 基本上任何东西都能掌握, 数学建模是对于算法有一些基本的要求, 对于模拟退火算法, 初看觉得很难, 但是简单调研一番后, 模拟退火算法是真的简单. 然后这期间也简单学了在数学建模培训赛题中使用模拟退火解决了经典的旅行商问题之类。

数学建模我们三个人做的是2020年的华为杯数学建模, 我们的题目是飞行器质心平衡供油策略优化. 我使用了python作为我们的编程与可视化语言. 因为, python 可以一套走完, 既可以完成数据从excel文档中的读取, 也可以使用python来进行图表的绘制.

两个问题的建模让我比较深刻

问题一是给出飞行器的俯仰角, 来计算整个飞机的质心变化曲线. 一个飞机有六个

油箱, 只有四个油箱可以进行出油, 每一时刻是有一个油箱可以对发动机进行供油. 虽然题目中给出了邮箱以长方体来进行建模但是由于油气是液体, 随着飞机的俯仰角会进行不规则体的变化, 我们分成四类情况进行讨论, 以长方形的方式来进行简化建模. 也就是说, 三棱柱, 四棱柱, 和五棱柱的情况. 分别求解出每一个邮箱的质心, 然后我们使用组合体质心求解公式对问题一进行了解答.

问题二, 给出了飞机的质心偏移数据, 我们要求解六个油箱的供油策略.

我们使用了理想执行偏移补偿算法. 就是下一个飞机的理想质心. 的偏移. 我们来进行六个油箱供油的遍历, 得到最能补偿偏移的向量使用这种方式决定油箱的供油选择.

数学建模让我个人成长了许多, 以前觉得模拟退火算法是比较难的算法, 因为随着这个算法, 一般会听到. 量子计算机, 这个是超过个人认知的东西, 但是经过个人的调研, 发现模拟退火算法可以说是, 智能算法中最简单的一种算法. 随着设定温度的下降整个系统达到稳定, 可以得到较优解. 也学习了蒙特卡洛算法, 以概率统计来进行结果的计算. 也学习了lingo等进行整数规划问题的求解.

感悟最多的是, 整个队伍还是比较难构建的. 比赛心态很重要, 虽然在最后一天比赛凌晨2点我才求解出问题四的解, 那个时候我的队友都有点要放弃的样子, 我还是对队友说给我1个小时就一定能写好. 个人觉得数学建模对于文档的撰写也很重要, 文档的图形与建模也是一项技术活儿.

13.1.0.4 之江天枢深度学习可视化项目

对于使用vue框架对于深度学习模型的高维向量分类效果进行了可视化. 数据分类效果的查看主要基于两种算法, 第一种是PCA(主成分分析, 就是将数据维度以权重的方式保留最高的), 另一种是TSNE(T分布和随机近邻嵌入)算法, 使用三种方式对数据进行可视化, 分为是 2位平面可视化数据点集, 三维空间可视化数据点集, 和 4维到8位的平行坐标可视化. 二维和4-8位采用的是d3.js对数据的可视化. 三维采用了echarts.js 进行数据可视化. 实现了动态数据变化. 动画效果是我做的两点之一, 因为D3.js 对于动画的支持比较靠后面, 因为D3.js 是支持定制化显示的. 但是对于动画的实现需要对D3.js 比较深刻的理解. 我做到了. 比如一个顶点从一个旧坐标到新坐标, 是采用连续的动画, 而不是顶点的直接跳跃. 数据类目以不同的三色进行表示, 清晰直观. 比如对于手写数字的可视化, 有10个颜色, 在后期模型训练好的时候, 在TSNE算法可以清晰的找到10个集合. 可以通过鼠标点击折现点, 如果某个分类集合中混入了别的颜色可以轻易找到, 进而看到原始的数据. 可以展示图片文字和音频. 增加了播放按钮, 随着模型的训练, 高维向量在后期, 几乎不会发生改变. 这样对于训练模型的研究者可以提前关闭模型的训练, 如果一个模型训练了很久高维数据还是分类不出来, 那么研究者可以简单判断模型构建出现了问题.

原本三维,个人是实现了, 图形学算法中的三维映射到二维的算法. 自己构建了MVP矩阵, 分别是模型矩阵, 视图矩阵, 和投影矩阵. 然后鼠标通过trackball来进行整个模型的旋转与缩放. 但是因为d3.js 只能操作svg, svg是一个比较古老的绘图方式, 性能并不高, Canvas 并不能通过d3.js 进行操作. 无奈之下, 使用了echarts.js现成的可视化方案.

第二个模块是媒体数据可视化模块, 使用element-ui进行制作, 可以显示图片, 音频, 文本等信息.

期间爬了很多坑, 比如echarts 传入的数据第一维度必须是标题, 不能直接是数据啥的. 文档也没说, 看了原码才知道, 对于echarts的源码进行了简单的debug. 有比较强的前端开发能力. 对于 VUE的 MVC 模型也比较熟悉, 数据驱动显示. 在我的项目中得到了淋漓尽致的利用.

项目中使用了less对于布局的简单使用, 使用async进行异步获取数据, 使用封装了axios的模块进行对于数据进行get获取。前端在第一次请求的时候获取了session_id, 然后之后都会携带有cookie对于身份的判断识别。

音频通过blob下载(里面好像使用了跨域, CustomAudio.vue就是实现这个的, 对于二进制文件内容的下载), 并且实现了音频组件的样式定时, 通过elementui连接audio中的事件, 通过更换UI设计师的icon使整个音频播放组件更加的美观.

数据的实时同步的实现, 通过前端实现了一个定时器, 然后每个正在展示的页面会定时发送数据请求, 然后更新数据的显示.

13.1.0.5 快乐慕商城电商后台项目介绍

(一期架构) 通过nginx反向代理, 使用tomcat作为服务端处理程序, 然后将session 存储到 guawa cache 中. 然后将数据存储的数据库中, 没有使用中间件处理, 单机架构, 导致系统的吞吐性能比较低. 在重置密码的时候需要带上一个有效的token来进行密码的重置

分成两个大方向来进行整个项目的构建. 分别是门户接口和后台接口.

门户接口主要分成6个方向, 支付, 用户, 购物车, 产品, 订单和收货地址.

后台接口主要分成5个方向, 产品, 统计, 用户, 品类和订单接口.

用户模块: 实现用户的注册. 登陆. 密码找回. 管理员登录等功能.

商品模块: 商品模块风味前台后来两个部分. 前台主要实现产品搜索. 动态排序类别. 商品详情展示等. 后台主要实现图片上传/商品上下架/增删改查商品等功能

购物车模块: 实现购物车中增添. 移除商品等功能.

收货地址模块: 实现用户收货地址的增删改查功能.

支付模块: 实现与支付宝的对接. 调通支付宝功能官方Demo.

虽然我们只完成了后端接口, 那我是如何测试的呢? 使用FeHelper 进行JSON美化, 使用 API Test (原名Restlet client) chrome 插件进行数据传输测试.

着重介绍user模块, 使用了guawa Cache 来进行token 的管理. 使用定时清理cache中的容量. 当我们使用问题来进行找回密码的时候, 如果问题的答案正确, 后台将会生成一个token, 然后下一个页面通过携带这个token来进行修改密码, 防止横向越权. Token 存储的是 key + value. 当回答问题正确的时候生成一个随机的UUID然后作为value, key就是token_字符串+用户名. 然后我们在登陆状态下修改密码的话, 我们需要检验用户名的旧密码, 防止出现横向越权的现象. 横向越权就是攻击者尝试访问与他拥有相同权限的用户的资源. 其中查看用户是否登陆通过httpsession 来查看用户是否已经登陆. 在登陆的时候设定session. 在注册密码的时候, 使用流读取配置文件存存储的MD5 盐值, 其中使用静态代码块, 来进行系统初始化的时候第一次读取盐值. 使用MD5 加盐的方式进行存储. 提升系统的安全性能.

还有一个我想介绍的是支付模块关于单边账问题, 很简单, 就是说我们记了, 支付宝没记, 或者支付宝记了, 我们系统没记。 1、每一笔交易一定要闭环, 即要么支付成功, 要么撤销交易, 一定不能有交易一直停留在等待用户付款的状态。 2、轮询+撤销的流程中, 如轮询的结果一直为未付款, 撤销一定要紧接着最后一次查询, 当中不能有时间间隔。 3、门店收银系统应该具备独立的手动查询功能作为兜底, 输入商户订单号（可从用户手机账单中获得）调用支付宝查询接口获得确切的支付状态。 4、当遇到网络超时和未知异常时, 参考异常处理流程正确处理, 对于每一笔交易或退款, 一定要得到确切的结果。 5、撤销接口调用成功后, 需要在收银台页面为收银员展示撤销成功的强提示文案, 且按实际业务情况引导收银员进行手工订单查询。 6、在上述基础上, 业务流程培训时应强调支付结果必须以商户端为准, 用户手机上的支付宝结果或账单只能做参考, 不能作为最终识别标准。如果商户未正确处理业务逻辑和业务流程培训, 存在潜在的风险, 商户自行承担因此而产生的所有损失。详见避免单边账说明。

(二期架构)

软件架构: Spring + SpringMVC + Mybatis + Nginx + Tomcat + Redis + Jedis + Lombok + Jackson + Guava Cache

使用横向扩展, 将session存储到redis中. 使用maven进行环境隔离, 分为 local dev 和prod 环境, lombok进行代码整洁性优化. 使用nginx进行负载均衡, 简单来说就是将请求分发到不同的tomcat中, 以权重的方式. redis使用一致性hash算法进行缓存分片部署. 封装了异常处理防止服务端关键信息泄露. 分布式锁进行了优化防止死锁.

一致性hash算法介绍: 简单来说生成多个虚拟节点, 以 $0 \leq i < 2^k - 1$ 的范围进行部署, 然后看顺时针映射到哪一个redis容器中.

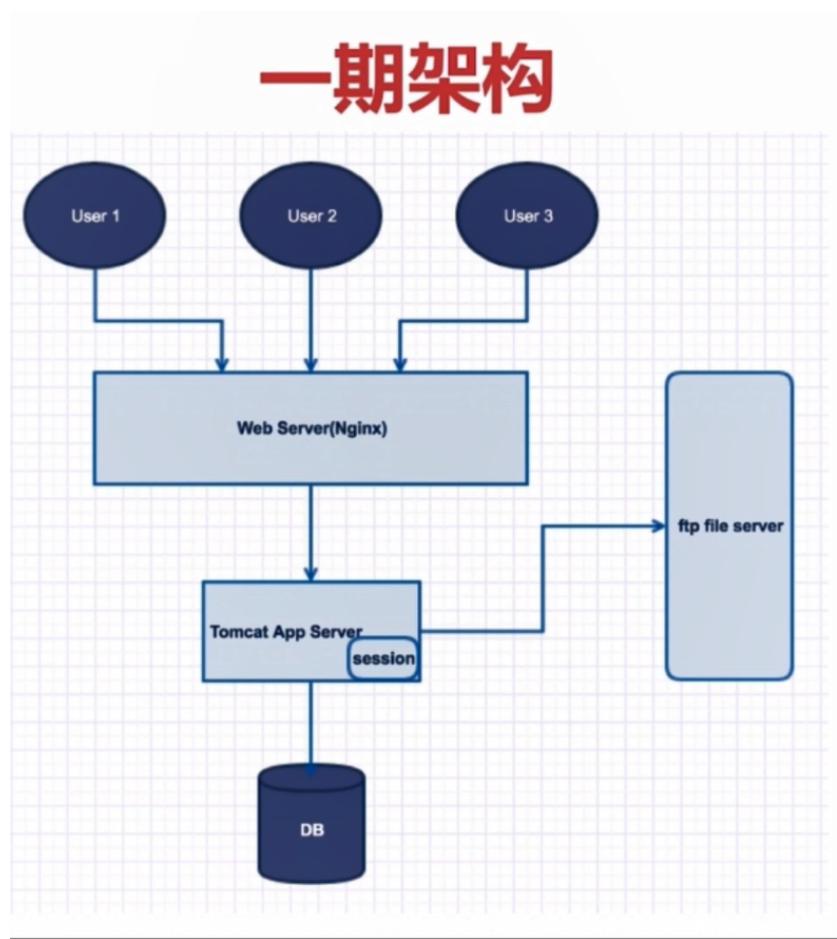


图 13-1 happymallOne

二期真架构

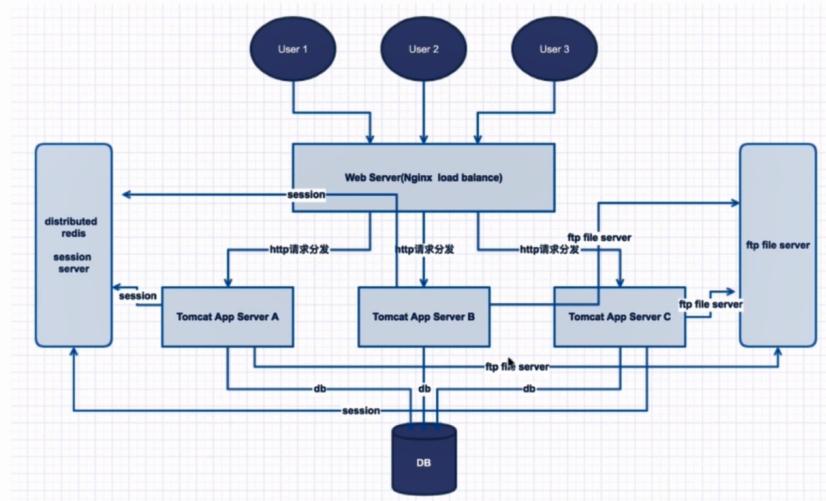


图 13-2 happymallTwo

Jackson: 主要通过objectMapper对象来实现json转string操作和string转json操作. 主要用在存储redis中的sessionid所存储的对象. 使用hash方式存储

slf4j+lockback: 进行日志管理 lockback 读取配置文件logback.xml进行解析

md5: 进行密码的加盐处理

Lombok: 用来简化代码的编写. 使用代码更加清晰

<mvc:interceptor> + handlerinterceptor: 使用拦截器来进行. 防止没有需要登陆的操作被使用. 如果是登陆操作使用单独的逻辑当然也可以使用相关的配置

用户模块, 登陆注册, 忘记密码, 其中忘记密码要重置的时候携带的token也是存放在redis中的

商品模块, 可以查看商品的详细信息, 使用了mybatis 的 pagehelper 分页器插件, 使用拦截器对sql进行limit处理

另一个我想介绍的是购物车模块. 购物车模块在展示购物车中选中的物品的时候, 使用DigDecimal类来进行数值的统计, 防止出现用为java, double有效位数导致的金额错误. 其中 bigDecimal类我们需要使用String类型的构造函数, 因为使用double类型的构造函数还是会出现精度问题.

13.1.0.6 秒杀项目介绍

mysql + reids + springboot + rockmq + nginx + @Transactional

1. 为什么要讲商品的库存表item_stock与商品表分开?

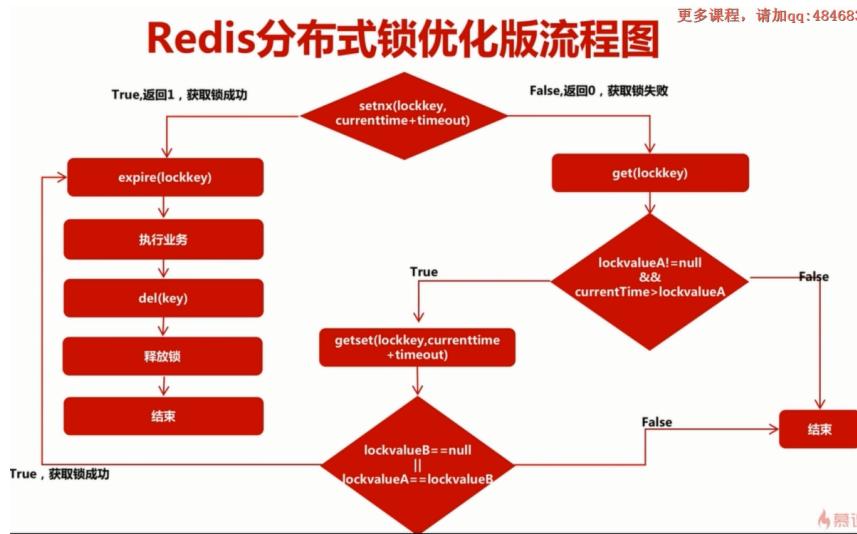


图 13-3 redislockdistribution

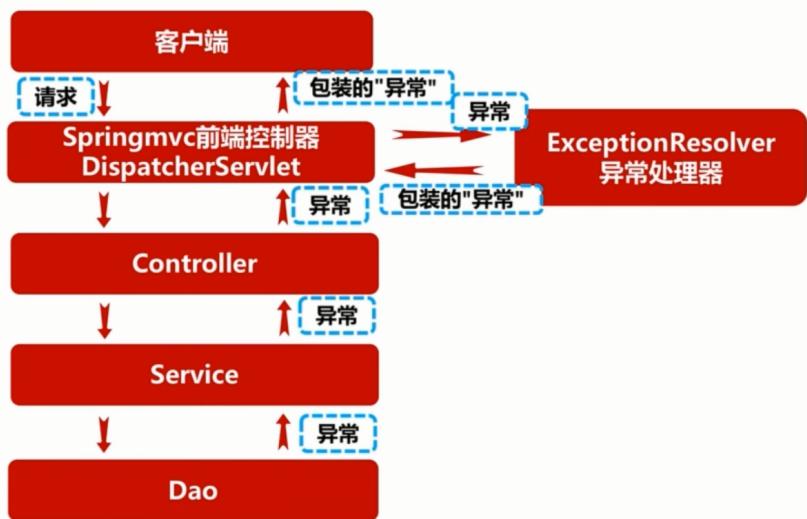


图 13-4 happymmallexception

库存操作非常耗时、性能，在商品交易过程中库存减，如果合并到item表中，每次会对对应行加行锁。如果分开库存表，虽然每次减库存过程还是会加行锁，但是可以将这张表拆到另一个数据库当中，分库分表，做效果的优化

```
@CrossOrigin(origins = "*",allowCredentials = "true")
```

我们的ajax看到这两个头部就认定对应的域名接收任何来自或不来自于本域的请求，简单来说，就是请求地址和域名请求都可以

2. tomcat 参数优化

增加线程数数量，最大工作线程数量增加到了200

keepAliveTimeOut: 设置30秒内没有请求则服务端自动断开keepalive链接

maxKeepAliveRequests: 当客户端发送超过10000个请求则自动断开keepalive链接

一定程度上减少了TCP三次握手的时间损耗。

3. 水平扩展

将前端静态资源直接放在nginx里面的html目录中。基于token的会话管理，前端存储到localStorage。后端存储到redis

4. 多级缓存

使用了三级缓存，第一级使用了guawacache存储了商品的信息，第二级 redis 第三季数据库使用序列化方式进行存储。商品列表可以达到Average time 150ms, 对应的Tps:2000/s, 对应的redis几乎没有任何压力，缓存机制从redis缓存加载到了jvm缓存之后，减少了多段的网络开销，并且完成了对应的内存访问输出结果，性能提升明显，但是数据更新之后缓存失效，还有JVM容量大小的限制；

5. 数据库存放入redis

会产生数据不一致的现象。

- (1) 活动发布同步库存进缓存
- (2) 下单交易减缓存库存
- (3) 异步消息扣减数据库内存

RocketMQ主要有Producer端，负责向Broker发送消息；Consumer端，多个consumer组成一个ConsumerGroup，每个消息会有一个Group里的consumer来消费；Broker由topic和MessageQueue组成，消息隶属于某个topic，一个topic可能由一个或多个topic管理

Producer连接NameServer发现broker1，会向topicA为主题的broker1投递消息，采用负载轮询向queue投递；

Consumer抓取负责的topicA，与queue建立长连接，当有消息时，唤醒，拉取对应的message，没有消息就等待，这种方式叫做长轮询

我们的解决方法就是异步消息的发送要在整个事务提交成功后再发送

6. 秒杀令牌

活动开始生成令牌

7. 设置秒杀大闸

就是一次释放的令牌大约是库存的5倍, + 验证码防止黄牛操作

8. 限流

使用令牌桶算法则是一个存放固定容量令牌的桶, 按照固定速率往桶里添加令牌。桶中存放的令牌数有最大上限, 超出之后就被丢弃或者拒绝。当流量或者网络请求到达时, 每个请求都要获取一个令牌, 如果能够获取到, 则直接处理, 并且令牌桶删除一个令牌。如果获取不同, 该请求就要被限流, 要么直接丢弃, 要么在缓冲区等待。

Guava RateLimiter

使用 RateLimiter的静态方法创建一个限流器, 设置每秒放置的令牌数为5个。返回的RateLimiter对象可以保证1秒内不会给超过5个令牌, 并且以固定速率进行放置, 达到平滑输出的效果。

9. 数据流

下单->流水库存->真正库存

10. 活动的开启与关闭, 通过一个内部的接口实现.

13.1.0.7 英文项目介绍

It's difficult to construct hexahedral meshes currently, this paper presents an interactive construction method of complex hexahedral mesh model based on volumetric subdivision. The user firstly needs to interactively construct the model skeleton, by placing cubes at the nodes of the skeleton structure, and performing interactive operations such as rotation, translation, and scaling of the node cubes. Then, through the connection between the nodes and the topological split operation, the initial control mesh can be constructed. Further, interpolatory Catmull-Clark volumetric subdivision method is used to generate hexahedral meshes with different resolutions; finally, the padding operation is used to eliminate the degraded elements at the boundary and improve the quality of the hexahedral mesh elements to obtain the final hexahedral mesh. The results of numerical example show that the method can easily and efficiently generate hexahedral meshes interactively, and eliminate the intermediate steps of generating volume meshes from surface meshes. It has some practical applications in finite element analysis, isogeometric analysis and geometric modeling in animation.

概念模型

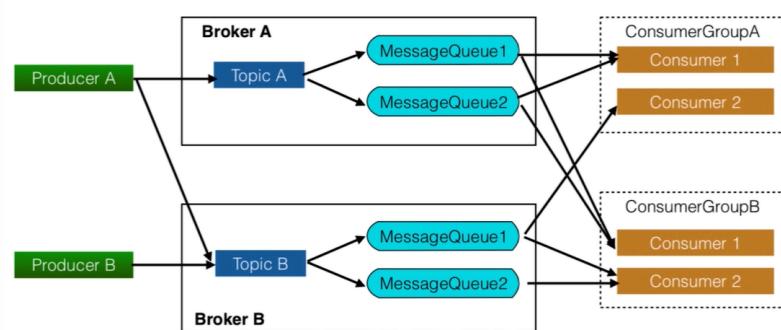


图 13-5 rockmq

14

游戏开发

► ...

14.1 知识点和方法论

14.1.0.1 渲染管线

图形渲染管线蛀牙包括两个功能: 1. 将物体3D坐标转变为屏幕空间2D坐标 2. 屏幕每个像素点进行着色

顶点数据的输入, 顶点着色器, 曲面细分过程, 几何着色器, 图元组装, 裁剪提出, 光栅化, 片段着色器以及混合测试.

15

自我

► ...

15.1 知识点和方法论

15.1.0.1 自我介绍

面试官好，我是杭州电子科技大学，计算机科学与技术专业的学生，来自浙江温州，研究生主要研究的方向是六面体网格生成。发表了一篇中文核心的论文。研究生期间参加过多种项目，典型的是参加过之江实验室的天枢深度学习可视化的项目，作为前端开发人员。同时也积极参与多种竞赛。参加过华为软件精英挑战赛和数学建模比赛。数学建模是国二。华为软件精英挑战赛今年拿过杭夏赛区的前64强。

我的兴趣爱好是跑步，参加过西湖毅行。

15.1.0.2 自我介绍英文版本

Hello everyone. I am liyongjie, a computer science and technology student from HangZhou Dianzi university. I grew up in WenZhou, but now I'm living in Hangzhou. I participated in several games including the Huawei Software elite Challenge, and mathematical modeling. More importantly I learnt how to communicate well with my team members, and how to division of labor Challenge. Then I partcipatd in several projects, including zhijiang Deep learning visualization. I worked one year and I was a software development. I like running in

my free time and I am so happy to be here and looking forward to working with you in the near future.

15.1.0.3 优缺点

优点是：坚持且不会轻易放弃，只要认为能做完可以做完，我会坚持做完它，比如数学建模，我们做的是比较慢，最后一天12点我们队伍还在做第四问，队友劝我放弃吧，先把前三问完成。我那个时候有思路了，然后我说等我2个小时，我一定可以做完，结果花了30多分钟就做好了，不过调试花了挺久的时间

缺点是：有一定拖延症，有的时候等到任务紧要的时候会效率比较高。

15.1.0.4 如何看待华为文化

狼性文化，胜则举杯相庆败则拼死相救；告诉每一个华为人，要追逐胜利不惧失败

15.1.0.5 遇到的困难以及解决办法？

首先遇到问题，根据问题去查找有没有相关的解决方案。

其次，如果没有解决方案，应该有相关知识点。根据相关知识点去搜相关论文。

如果还没有解决方案，可以和朋友和老板沟通交流一下。有的时候在沟通交流的时候就会出现解决方案。

15.1.0.6 未来3 5年的规划和职业目标是什么？

前两年，好好学习掌握华为的技术，后面几年争取输出技术。

职业目标是：能够独立的带领团队开发项目。

15.1.0.7 加入不同项目组认为自己的板块是最重要的，没有及时给你配合或者你遇到配合上，进度上的问题如何解决？

一方面对自己上下游的工作内容要有一定的熟悉，如果是上游出现了问题，那只能把问题反馈给上游，术业有专攻。

二是，在别人有空的时候要经常去找对方沟通交流，如果是要配合的一些小事，如果能自己干的话就自己干了。

15.1.0.8 反问

部门的发展方向是什么?

什么时候有通知

15.1.0.9 为什么读研

1. 一方面想要完成自己的小小愿望, 在大学期间拿一个奖项. 本科也参加过电子设计竞赛, 但是这种比赛比较吃实验室的资源. 学计算机的话, 比如数学建模, 只要一台电脑就可以了.
2. 工资比较低, 感觉不如读研, 然后找个好工作.

15.1.0.10 用英语简单聊

16

常用算法题以及思路

► ...

16.1 知识点和方法论

16.1.0.1 <https://leetcode-cn.com/problems/longest-increasing-subsequence/submissions/>

思路: 两重for循环

```
1 if(n[i] > n[j]) {  
2     dp[i] = max(dp[i], dp[j]+1);  
3 }
```

16.2 智力题

16.2.0.1 一根金条，需要付给工人，工人工作七天，需要每天给他一段1/7的金条，如何保证切两刀就完成任务。

https://blog.csdn.net/qq_43264029/article/details/107270544

16.2.0.2 浅谈一下1000瓶水有关思路(图解)与用java代码具体解决方案

<https://blog.csdn.net/sunkun2013/article/details/12035071>

17

系统设计题

► ...

17.1 知识点和方法论

17.1.0.1 分布式ID生成器

使用UUID 4个字节表示的Unix timestamp, 3个字节表示的机器的ID 2个字节表示的进程ID 3个字节表示的计数器

SnowFlake 算法，是 Twitter 开源的分布式 id 生成算法。其核心思想就是：使用一个 64 bit 的 long 型的数字作为全局唯一 id。在分布式系统中的应用十分广泛，且 ID 引入了时间戳，基本上保持自增的，后面的代码中有详细的注解。

给大家举个例子吧，比如下面那个 64 bit 的 long 型数字：

第一个部分，是 1 个 bit: 0，这个是无意义的。

第二个部分是 41 个 bit: 表示的是时间戳。

第三个部分是 5 个 bit: 表示的是机房 id, 10001。

第四个部分是 5 个 bit: 表示的是机器 id, 1 1001。

第五个部分是 12 个 bit: 表示的序号，就是某个机房某台机器上这一毫秒内同时生成的 id 的序号，0000 00000000。

18

bash

► ...

18.1 知识点和方法论

18.1.0.1 awk 命令

% 是用来格式化字符串的 \$ 是用来取出变量的

```
ps -ef|grep "synergy"|awk'print$2'|xargskill -9
```

18.1.0.2 找出那些js文件没有设置好权限

find + perm + mode

```
find . -name "*.js" -perm 644 -print
```


19

数量关系总览

► ...

19.1 知识点和方法论

19.1.1 分数

$$\frac{1}{2} = 0.5$$

$$\frac{1}{3} = 0.33$$

$$\frac{1}{4} = 0.25$$

$$\frac{1}{5} = 0.2$$

$$\frac{1}{6} = 0.167$$

$$\frac{1}{7} = 0.142$$

$$\frac{1}{8} = 0.125$$

$$\frac{1}{9} = 0.111$$

$$\frac{1}{125} = 0.008$$

19.1.2 圆

$$\tan 15 = 0.268$$

19.1.3 空瓶换酒

M-1个空瓶可以换酒

19.1.4 路程

$$60 \text{ km/h} = 60 / 3.6 =$$

19.1.5 平年, 闰年

平年星期 + 2

闰年星期 + 1

19.1.6 倍数

$$2^2 = 4; 3^2 = 9; 4^2 = 16; 5^2 = 25; 6^2 = 36; 7^2 = 49; 8^2 = 64; 9^2 = 81; 11^2 = 121; 12^2 = 144; 13^2 = 169; 14^2 = 196$$

19.1.7 奇偶特性

简单来说就是从奇偶特性推导出答案

1. 奇数 \pm 奇数 = 偶数; 偶数 \pm 偶数 = 偶数;
- 偶数 \pm 奇数 = 奇数; 奇数 \pm 偶数 = 奇数;
2. 奇数 \times 奇数 = 奇数; 奇数 \times 偶数 = 偶数; 偶数 \times 奇数 = 偶数; 偶数 \times 偶数 = 偶数;

19.1.8 倍数特性

1. 3, 9 整除判定基本法则

一个数能被3整除, 当且仅当其各位数字之和能被3整除;

一个数能被9整除, 当且仅当其各位数字之后能被9整除;

2. 2, 4, 8, 5, 25, 125 整除判定基本法则

一个数能被2(或者5)整除, 当且仅当末一位数字能被2(或者5)整除;

一个数能被4(或者25)整除, 当且仅当末两位数字能被4(或者25)整除;
 一个数能被8(或者125)整除, 当且仅当末三位数字能被8(或者125)整除;

19.1.9 方程法

设未知数时候, 一般采取设小不设大
 加速

$$\frac{a}{c} = \frac{b}{d} = \frac{a+b}{c+d} = \frac{a-b}{c-d}$$

19.1.10 赋值法

一般对效率, 成本, 进价等赋值时, 昌吉和比例关系赋值简单数, 数字要尽可能地便于计算和化简, 如1, 2, 60, 100等.

赋0法, $x+y+z$ 一定, 那么直接设其中一个值为0

19.1.11 等距离平均速度公式

$$\text{等距离平均速度} = \frac{2v_1v_2}{v_1 + v_2}$$

19.1.12 利润率问题

$$\text{利润率} = \text{利润} \div \text{进价}$$

19.1.13 溶质问题

$$\text{浓度} = \frac{\text{溶质质量}}{\text{溶液质量}}$$

19.1.14 排列组合

$$A_{10}^3 = 10 \times 9 \times 8$$

$$C_{10}^3 = \frac{10 \times 9 \times 8}{3 \times 2 \times 1}$$

19.1.15 几何

$$\text{球的表面积} = 4\pi R^2$$

$$\text{球的体积} = \frac{4}{3}\pi R^3$$

19.1.16 周期问题

每n天的一个周期, 为n天

每隔n天的一个周期是(n+1)天

19.1.17 全错误排列

主体为4辆车, 要求所有车都不得停在原来的车位中, 则一共有多少种不同的停放方式.

记住结论 $D_1 = 0, D_2 = 1, D_3 = 2, D_4 = 9$

19.1.18 三集合容斥原理

$$A + B + C - A \cap B - A \cap C - B \cap C + A \cap B \cap C = \text{总数} - A, B, C, \text{均不满足的个数}$$

$$A + B + C = a + 2b + 3C$$

a,b,c分别表示满足一个条件的数量和满足2个条件的数量和满足三个条件的数量

$$A + B + C - b - 2c = \text{总数} - A, B, C \text{均不满足的个数}$$

19.1.19 追及问题

300m环形跑道上从同一点触发同向而行, 每追上一次则多跑一圈. A每次追上B后都减速0.5m/s, 当A从6m/s减速到3m/s, 根据

$$S = 300 = (V_A - V_B) \times T$$

共同跑的路程是990m

19.1.20 代入排除法

1. 尾数法加速排除

2. 代入排除不是无脑代入, 需要有所选择. 一般题目问最大, 那我们就考虑从最大的开始代入, 问最小就从最小开始代入.

19.2 数量关系补充

19.2.1 蒙题大法

1. 问最大, 蒙次大
2. 问最小, 蒙次小或最大
3. 3+1蒙题:三项等差/等比数+1项特殊项, 蒙接近特殊项的一项
4. 迷惑项蒙题, 看问题, 和差倍比关系
5. 找共性蒙题
6. 常识蒙题

19.2.2 例题: 圆桌公式

2个大人带4个小孩去坐旋转木马, 问不相邻的概率

$$A_{n-1}^{n-1}$$

$$A_3^3 \times A_2^4 \div A_5^5 = 72 \div 120$$

19.2.3 例题: 周期公式

网管小刘负责A,B,C三个机房的巡查工作, A,B,C分别需要每隔2,4,7天巡检一次, 3月1日, 小刘巡检了三个机房, 问他在整个3月有几天不用做机房的巡检工作

TIPS: 每隔N天=每(N+1)天

在3月份身下30天中, A需要 $\frac{30}{3} = 10$, B需要 $\frac{30}{5} = 6$, C $\frac{30}{8} = 3$

A 和 B $\frac{30}{15} = 2$

B 和 C $\frac{30}{40} = 0$

A 和 C $\frac{30}{24} = 1$

A 和 B 和 C $10 + 6 + 3 - 2 - 1 = 16$ 天, 休息 14天

19.2.4 例题: 单端触发相遇问题

$$2nS = (v_1 + v_2)t$$

n 相遇次数, t 时间

小王和小李沿着绿岛往返运动, 绿道总长度为3km. 小王每小时走2km; 小李每小时跑4km. 如果两人同时从绿道的一端触发, 则当两人第7次相遇时, 距离触发点(多少公里)

20

判断推理总览

► ...

20.1 知识点和方法论

20.1.1 推理

1. 如果要

如果去新疆, 就要游吐鲁番和喀纳斯

=> 去新疆 -> 游吐鲁番和喀纳斯

2. 只有才 (后推前)

只有与小李同游, 小张才会游吐鲁番或天池

=> 游吐鲁番或天池 -> 与小李同游

3. 否后比否前

游吐鲁番或天池 -> 与小李同游 -> 与小李做约定 -> 小李这个夏天一定有时间

否后比否前, 推出 -(游吐鲁番或天池), 根据德摩根定律, -吐鲁番且-天池. 推出小张没有去新疆

4. 德摩根定律

$-(A \text{ 或 } B) = -A \text{ 且 } -B$

5. 或或

否一, 推一

只有拥有强大的科技创新能力, 才能增强综合国力, 才能提高国际竞争力. = 国际竞争力 \rightarrow 强大的科技创新能力

我国或者国际竞争力得不到提高, 或者拥有强的科技创新能力. = -国际竞争力或有强的科技创新能力 \Rightarrow 国际竞争力 \rightarrow 强大的科技创新能力

5. 既是 ... 更是 ...

贫困群众既是脱贫攻坚的扶贫对象更是脱贫致富的主体力量.

贫困群众 \rightarrow 脱贫攻坚的扶贫对象且脱贫致富的主体力量

6. 所有... 都

前对后

所有脱贫攻坚的扶贫对象都是脱贫致富的主体力量.

20.1.2 图形推理

1. 思路

- a. 各图形构成相同, 一般考察位置规律
- b. 各图形构成相似, 一般考察样式规律
- c. 各图形构成不同, 一般考察属性, 数量及其他特殊规律
- d. 圆相切和相交, 可以考虑一笔画

20.1.3 九宫格图推

1. 如果平移元素只在九宫格或十六宫格图形的最外圈出现, 有限考虑元素在最外圈按顺时针方向平移.
2. 如果平移元素出现在非最外圈位置, 优先考虑直线方向平移.

20.1.4 翻转

20.1.5 样式规律

题型特征: 图形元素组成相似

1. 加减同异

相加: 将两图形中所有元素拼合成一幅新图形

相减: 当第一幅图的元素或线条完全包含第二幅图时, 两图相减的结果, 就是第一幅图去掉第二幅图所有元素之后的图形

求同: 将两图形中所有不同的元素去掉, 只留下相同的部分, 形成一幅新图形.

求异: 将两图形中所有相同元素去掉, 只留下各自不同的部分形成一幅新图形.

2. 缺啥补啥

当各图形组成相似, 且某些元素或特征不止一次出现在各图形中时, 优先考虑缺啥补啥.

3. 叠加运算

一般要列出四个公式

$$\text{白} + \text{白} = ?$$

$$\text{黑} + \text{黑} = ?$$

$$\text{黑} + \text{白} =$$

$$\text{白} + \text{黑} =$$

20.1.6 属性规律

题型特征: 图形元素组成不同

1. 对称性

- a. 轴对称
- b. 中心对称
- c. 既是轴对称也是中心对称
- d. 对称轴的方向和数量

2. 曲直性

- a. 曲: 图形只由曲线构成
- b. 直: 图形只由直线构成
- c. 曲+直: 图形由曲线和直线共同构成

3. 开闭性

- a. 开放图形, 不行不包含任何封闭空间, 即没有窟窿
- b. 封闭图形, 图形包含封闭空间, 即有窟窿

20.1.7 数量规律

题型特征: 图形元素组成不同, 且无明显属性规律.

1. 点

切点, 直线和曲线的交点.

2. 线

当图形中出现多边形或单独的一条直线时, 优先考虑数直线; 当图形中出现较多曲线时, 优先考虑数曲线.

3. 数笔画

a. 连通图的笔画数 = 奇点数 / 2;

奇点: 若以一个点为起点, 延伸出的线条数为奇数, 则该店为奇点

b. 长考笔画规律的特征图形: 五角星, 月亮, "日", "田", 当看到题目中出现以上特征图形时, 可优先考虑数笔画.

4. 数面

5. 数元素

元素的个数, 种类, 部分数

6. 数角

20.1.8 特殊规律

图形间的规律: 相离, 相压

相交

a. 相交于面, 相交于点, 相交于边

20.1.9 空间重构

1. 六面体

在立体图智能看到一个对面, 不能看到两个对面

通过在公共边顺时针画一圈

三个面相邻公共点不变

2. 截面图

3. 三视图

20.1.10 定义判断

1. 关键词

主客体

关联词

方式+目的端口

a. 按照 / 通过 / 采用 / 利用的方式方法 / 办法/ 依据 / 手段

b. 以 / 达到 / 实现

2. 同构选项进行排除

20.1.11 类比推理

1. 语义关系
 - a. 近义词
 - b. 反义词
 - c. 比喻词
2. 全同关系
一年四季, 春夏秋冬.
3. 包容关系
 - a. 种属关系
 - b. 组成关系
4. 并列关系
 - a. 容斥关系
 - b. 非容斥关系
5. 交叉关系
6. 对应关系
 - a. 配套使用, 牙刷 & 牙膏
 - b. 物品与原材料, 制作工艺.
 - c. 物品与功能
 - d. 属性关系
 - e. 因果关系
7. 语法关系
 - a. 词性: 名词, 动词, 形容词
 - b. 顺序: 题干和选项用同样的顺序造句.

20.1.12 翻译推理

20.1.12.1 前推后

1. 如果 ... 就 ...
2. 只要 ... 就 ...
3. 所有 ... 都 ...
4. ... 是 ... 的充分条件
5. ... 就/则/都/一定...

20.1.12.2 后推前

1. 只有 ... 才..
2. 不 ... 不 ...
3. ... 才 ...
4. 除非 ... 否则不 ...
要么不回答, 除非含糊不清
回答 -> 含糊不清
5. ... 是 ... 的必要条件
除非田宇去海洋社区, 否则王栋不去文明社区
王栋去文明社区 -> 田宇去海洋社区

20.1.12.3 推理规则-逆否等价

$$\begin{aligned} a \rightarrow b \\ \Rightarrow \\ \neg b \rightarrow \neg a \end{aligned}$$

20.1.12.4 传递规则

20.1.12.5 tricks

谁必不可少, 谁在箭头后面
德摩根定律
 $\neg(a \text{ 且 } b) = \neg a \text{ 或 } \neg b$

20.1.13 排列组合

代入排除不是无脑代入
最大信息法
列表

20.1.14 加强与削弱

20.1.14.1 削弱

否定论点
拆桥

论点: 有1就有2

削弱方式1 – 否定论点: 有1同时没有2

削弱方式2 – 可能性拆桥: 没有1同时有2

方式1比方式2削弱力度强

削弱方式3 否定论据

削弱方式4 因果倒置与另有他因

1. 支持方: 1是2的原因; 反对方: 1不是而的原因, 三是二的原因

2. 削弱反对方: 1是3的原因, 故而1也是2的原因.

否论点强于否论据

20.1.14.2 加强

加强方式1 搭桥

加强方式2 必要条件

加强方式3 解释与举例

20.1.14.3 日常结论

1. 三不选

a. 存在逻辑错误的选项一定不选

b. 无中生有的选项一定不选

c. 偷换概念的选项一定不选

2. 两慎选

a. 概念扩大

b. 有敏感词的慎选, 绝对

3. 一定选

可能

20.2 判断推理

20.2.1 例题

如果去新疆, 就要游吐鲁番和喀纳斯

=> 去新疆 -> 游吐鲁番和喀纳斯

只有与小李同游, 小张才会游吐鲁番或天池

=> 游吐鲁番或天池 -> 与小李同游

如果与小李同游, 小张一定要与小李做约定

=> 与小李同游 -> 与小李做约定

如果小张与小李做约定, 则小李这个夏天一定有时间

=> 小张与小李做约定 -> 小李这个夏天一定有时间

遗憾的是, 小李没去

总路线

游吐鲁番或天池 -> 与小李同游 -> 与小李做约定 -> 小李这个夏天一定有时间

否后比否前, 推出 -(游吐鲁番或天池), 根据德摩根定律, -吐鲁番且-天池. 推出小张没有去新疆

20.2.2 正方形

1. 对面
2. 顺时针画圈

21

常识

► ...

21.1 知识点和方法论

21.2 常识总览

1. 习近平新时代中国特色社会主义思想, 其核心要义是, 坚持和发展中国特色社会主义
2. 党的十九大报告, 发展是解决我国一切问题的基础和关键,
3. 带领人民创造美好生活是我们党始终不渝的奋斗目标
4. 核力量是维护国家主管和安全的战略基石
5. 相顾无相识, 长歌怀采薇, 大家相对无言彼此互不相识, 作者长啸高歌真想隐居在山冈.
6. 特赦令由国家主席签发, 全国人大常委会有权决定特赦
7. 洞庭湖在湖南省
8. 党的十九大做出 2020年实现机械化, 2035年实现国防和军队现代化, 本世纪中叶实现人民军队建成世界一流军队.
9. 钢铁业和化工业属于资本密集型产业
10. 腐乳没有用到乳酸菌
11. 红薯明代才有
12. 菡萏 – handan 指荷花

13. 蜘蛛不是昆虫
14. 党的十八届三中全会首次提出"推进国家治理体系和治理能力现代化"这个重大命题, 并把"晚上和发展中国特色社会主义制度, 推进国家治理体系和治理能力现代化"确定为全面深化改革的总目标.
15. 1929年, 召开古田会议, 加强党对军队的领导
16. 1947年, 西柏坡全国土地会议, 实施了<中国土地法大纲>
17. 行政复议法, 有下列情形之一的, 公民, 法人或者其他组织可以依照本法申请行政复议: 对行政机关作出的关于确认土地, 矿藏, 水流, 森林, 山岭, 草原, 荒地, 谈吐, 海域等自然资源的所有权或者使用权不服的
18. 中国共产党在中国革命中战胜敌人的三个法宝是统一战线, 武装斗争, 党的建设.
19. 街道办事处是区人民政府的派出机关, 居民委员会是居民自我管理, 自我教育, 自我服务的基层群众性自治组织.
20. 南水北调工程连接了长江, 黄河, 淮河, 海河.
21. 高质量发展的第一要义是发展
22. 完成了消除绝对贫困的任务
23. 以党的政治建设为中心, 而不是思想工作
24. 三湾改编, 从组织上确立了党对军队的领导
25. 古田会议, 确立了马克思主义建党建军原则, 确立了军队政治工作的方针, 原则和制度.
26. 党的十九大, 把中央军事委员会实行主席负责制写入党章.
27. 爱国统一战线是重要法宝
28. 故意犯罪, 应予以开出, 而非撤职
29. 深海一号是母船, 奋斗者号是潜水器.

21.2.1 常识蒙题

<https://www.bilibili.com/video/BV1bK411K7AK?p=33> 李铁老师

21.3 常识

21.3.1 关键语句

1. 十四五时期推动高质量发展, 必须立足新发展阶段, 贯彻新发展理念, 构建新发展格局.
2. 严格执行市场准入负面清单, 普遍落实"非禁即入"

3. 嫦娥五号返回器携带月球样品, 是2020年12月17号发生的.
4. 组织处理, 是指党组织对违规违纪违法, 失职失责失范的领导干部采取的岗位, 职务, 职级调整措施.
5. 货币(M1)=流通中的现金+银行储蓄(活期), 货币(M2) = M1 + 定期存款
6. 张三, 在高考期间鸣笛, 对罚款行为提起复议时, 可附带要求审查省政府的规定.
7. 张三若对县公安局的罚款处罚不服的, 可以向上一级主管部门即市公安局, 或者本级政府即县政府申请行政复议.
8. 张三若对此罚款不服, 可不经过复议, 直接向人民法院提起诉讼.
9. 张三可以自知道该具体行政行为之日起六十日内提出行政复议申请; 但是法律规定的申请期限超过六十日的除外.
10. 中共二大通过了第一份党章

22

资料分析总览

► ...

22.1 知识点和方法论

22.1.1 速算技巧

分子和分母小

$$\frac{1 + 8.6\%}{1 + 12.5\%} = 1 - 4\%$$

特殊分数

$$\frac{\text{现期量}}{1 + \frac{1}{8}} * \frac{1}{8} = \frac{\text{现期量}}{9}$$

$$\frac{\text{现期量}}{1 - \frac{1}{8}} * \frac{1}{8} = \frac{\text{现期量}}{7}$$

22.1.2 增长量比较

Ar1 ?= Br2

22.1.3 术语

同比, 是今天6月对比去年6月.

环比, 是今天6月对比去年5月.

22.1.4 特定历史时期

九五计划 1996 - 2000年

十三五计划 2016 - 2020年

22.1.5 结构阅读法

略读资料, 详读结构.

22.1.6 间隔增长率

2013年相对于2011年增长了

$$r_{\cdot} = r_1 + r_2 + r_1 \times r_2$$

22.1.7 年均增长率

$$\text{基期量} \times (1 + \text{年均增长率})^n = \text{现期量}$$

22.1.8 基期比重

部分/整个

2018年 A , B , a, b

2017年 A/(1+a) , B /(1+b)

2017年比重 A/B*((1+b)/(1+a))

22.1.9 平均数

平均每(分母)

22.1.10 平均值增长率

2017 A a

2017 B b

2018 A(1+a)

2018 B(1+b)

$$r = \left(\frac{A(1+a)}{B(1+b)} - \frac{A}{B} \right) \div \frac{A}{B} = \frac{1+a}{1+b} - 1 = \frac{a-b}{1+b}$$

22.1.11 年均增长量

$$\text{年均增长量} = \frac{\text{现期量}(2020)-\text{基期量}(2011)}{\text{年份差}(2020-2011)}$$

22.1.12 多多少倍数

n倍 - 1 个

22.1.13 基期倍数公式

$$\text{OffP} = \frac{A}{1+a} \div \frac{B}{1+b} = \frac{A}{B} \times \frac{1+b}{1+a}$$

A B 表示现期量

a,b 表示对应增长率

22.1.14 两期增长比重

$$\text{两期比重差} = \frac{A}{B} - \frac{A}{B} \times \frac{1+b}{1+a} = \frac{A}{B} \times \frac{a-b}{1+a}$$

2018 , A, B, a, b 2017, A/(1+a), B/(1+b)

A 部分, B 整理, 一般选择小的

22.1.15 平均增长量

$$\text{平均数增长量} = \text{现期平均数} - \text{基期平均数} = \frac{A}{B} - \frac{A}{B} \times \frac{1+b}{1+a} = \frac{A}{B} \times \frac{a-b}{1+a}$$

其中A表示现期总量, B表示现期个数.

22.1.16 两期比重差

$$\text{两期比重差} = \frac{A}{B} \times \frac{a-b}{1+a}$$

22.1.17 平均增长率

$$\text{平均数增长率} = \frac{\text{现期平均数} - \text{基期平均数}}{\text{基期平均数}} = \frac{\frac{A}{B} - \frac{A}{B} \times \frac{1+b}{1+a}}{\frac{A}{B} \times \frac{1+b}{1+a}} = \frac{a - b}{1 + b}$$

其中A表示现期总量, B表示现期个数.

22.1.18 加速计算

1. 截位法, 简单来说就是被除数, 进行保留2-3位进行计算加速.

看选项差距, 选项差距大, 保留2位, 选项差距小保留三位.

2. 估算公式

$$\frac{A}{1 \pm r} \doteq A(1 \mp r)$$

当 $r < 5\%$

23

言语总览

► ...

23.1 知识点和方法论

23.1.1 中心理解题

注意:

关联词, 主题词, 程度词.

转折之后的新观点为本文的重点.

因果关系之后的结论是重点

必要条件, 只有...才之间的中点

解决问题的对策是重点

反面论证, 简而言之, 就是对策

并列关系, 答案中要都有

中心句, 常见于段首和段尾

23.1.2 细节题的容易错误的点

1. 无中生有
2. 偷换概念
3. 偷换时态
4. 程度轻重

23.1.3 排序

23.1.3.1 确定首句

1. 定义排第一句
2. 背景引入
3. 提出观点

23.1.3.2 确定尾句

引入结论和提出对策的语句通常适合作为尾句

典型标志词: 因此, 所以, 看来, 这+应该, 需要

23.1.4 篇章阅读题

1. 明确阅读顺序, 先根据题干判断题型优先级
2. 利用题干关键词定位原文
3. 合理调整做题顺序, 不一定位, 细节理解题可以考虑放弃
4. 依托重点提示把握文段结构
5. 把握做题时间

23.1.5 定义

妄言: 胡言乱语, 瞎编

推脱: 推卸责任

推托: 婉言拒绝

负担: 一般指消极的事物

23.2 申论总览

23.2.1 规范词梳理

1. 发展特色产业
2. 提高竞争力
3. 提高抗风险能力
4. 稳定就业
5. 按需设岗

6. 改善办学条件

23.2.2 tricks

不懂题干在说什么摘抄对策就完美了

好的影响(例子) + 坏的影响(例子) + 观点(主题)

举例子

化大为小, 将很抽象的概念用很多的小概念进行概括

题目: 动词 + 主题词

23.3 言语

23.3.1 tricks

用词语的感情色彩来判断.

23.3.2 成语积累 & 词语积累

方兴未艾: 多形容新生事物正在蓬勃发展.

独善其身: 指不做官, 就搞好自身的修养. 现在也指只顾自己, 缺乏集体精神.

以邻为壑: 指那邻国当做大水坑, 把本国的洪水排泄到那里去, 比如吧困难或灾祸退给别人.

手足无措: 形容举动慌张, 侧重于强调处于非常窘迫的外在状态.

亦庄亦谐: 指讲话或文章既庄重正派, 又幽默活泼.

面面俱到: 指各方面都要照顾到, 没有遗漏

等量齐观: 有差别的事物同等看待

缘木求鱼: 比喻方法不对头, 不可能达到目的.

羁绊: 起到消极作用.

羔羊跪乳: 比如乌鸦反哺, 孝敬长辈

言之凿凿: 说明有证据

彩衣娱亲: 比喻孝顺.

墨菲定理: 如果事情有变坏的可能, 不管这个可能性有多小, 它总会发生.

苦心孤诣: 指莎菲苦心地钻研, 到了别人所达不到的至高境界.

青灯黄卷: 形容清苦的攻读生活.

一往而深: 对人或事物倾注了恒盛的感情, 向往而不能克制

久久为功: 持之以恒, 锲而不舍, 驰而不息

斑驳陆离: 形容颜色杂乱的样子

坐而论道: 口头说说, 不见行动, 侧重空谈

老调重弹: 说过多次的理论, 主张重新搬出来, 不能体现出"没有接受教训"的含义, 与文意不符, 排除.

自成一体: 在书法, 绘画等方面具有独创风格, 能自成体系.

24

流体力学

► ...

24.1 知识点和方法论

24.1.1 流体力学需要遵守的三个物理学原理

1. 质量守恒定律
2. 牛顿第二定律(力 = 质量 × 加速度)
3. 能量守恒定律

24.1.2 流体力学基本控制方程

1. 连续性方程
2. 动量方程
3. 能量方程

24.1.3 流体力学流动模型

1. 固定的有限控制体, 直接导出守恒型积分方程
2. 移动的有限控制体, 直接导出非守恒型积分方程
3. 固定的无穷小控制体, 直接导出守恒型偏微分方程
4. 移动的无穷小控制体(移动的流动微团), 直接导出非守恒型偏微分方程

24.1.4 条件

1. 边界条件
2. 无粘流
3. 粘性流

24.1.5 速度向量场

$$\vec{V} = u\vec{i} + v\vec{j} + w\vec{k}$$

速度分量

$$u = u(x, y, z, t)$$

$$v = v(x, y, z, t)$$

$$w = w(x, y, z, t)$$

24.1.6 标量密度场

$$\rho = \rho(x_1, y_1, z_1, t_1)$$

24.1.7 物质导数

$$\lim_{t_2 \rightarrow t_1} \frac{\rho_2 - \rho_1}{t_2 - t_1} = \frac{D\rho}{Dt}$$

密度的物质导数就是, 流体微团在空间运动时, 其密度的时间变化率. 它和偏导数 $\partial\rho/\partial t$ 不同, 偏导数表示的是在固定位置点的变量率.

$$\frac{D\rho}{Dt} = \frac{\partial\rho}{\partial t} + u \frac{\partial\rho}{\partial x} + v \frac{\partial\rho}{\partial y} + w \frac{\partial\rho}{\partial z}$$

$$\text{笛卡尔坐标系下向量算子 } \nabla = \vec{i} \frac{\partial}{\partial x} + \vec{j} \frac{\partial}{\partial y} + \vec{k} \frac{\partial}{\partial z}$$

$$\text{物质导数则为 } \frac{D}{Dt} = \frac{\partial}{\partial t} + \vec{V} \cdot \nabla$$

$\frac{\partial}{\partial t}$ 叫做当地导数, 它在物理上是固定点出的时间变化率.

$\vec{V} \cdot \nabla$ 叫做迁移导数, 它在物理上表示由于流体微团从刘长忠的一点运动到另一点, 流畅度额空间不均匀性而引起的时间变化率.

物质导数和时间的全导数是相同的.

24.1.8 速度散度

$$\nabla \cdot \vec{V} = \frac{1}{\delta V} \frac{D(\delta V)}{Dt}$$

右边就是速度散度, 左边就是速度散度的物理意义, 即 $\nabla \cdot \vec{V}$ 是每单位体积运动着的流体微团, 体积相对变化的时间变化率.

24.1.9 空间位置固定的有限控制体模型

通过控制面S流出控制体的净质量流量(B) = 控制体内质量减少的时间变化率(C)

通过面积dS的质量流量微元为 $\rho \vec{V}_n dS = \rho \vec{V} \cdot d\vec{S}$

$d\vec{S}$ 的方向总是指向控制体外. 当 \vec{V} 也指向控制体外, 表示质量流量在物理上是离开控制体的, 也就是流出. 因此,

$$B = \iint_S \rho V \cdot d\vec{S}$$

$$C = -\frac{\partial}{\partial t} \iiint_V \rho dV$$

$$B + C = 0$$

是连续性方程的积分形式. 控制体有限的体积就是方程具有积分形式的原因.

24.1.10 随流体运动的有限控制体模型

该控制体优势有相同的, 可辨认的质量微团组成. 也就是说, 这种运动的控制体具有固定不变的质量.

有限控制体的总质量 =

$$m = \iiint_V \rho dV$$

整个控制体的质量的物质导数为0

$$\frac{D}{Dt} \iiint_V \rho dV = 0$$

24.1.11 空间位置固定的无穷小微团模型

我们定义净流出量为正

那么我们有, x 方向的净流出量

$$[\rho u + \frac{\partial(\rho u)}{\partial x}] dy dz - (\rho u) dy dz = \frac{\partial(\rho u)}{\partial x} dx dy dz$$

y 方向的净流出量

$$[\rho v + \frac{\partial(\rho v)}{\partial y}] dx dz - (\rho v) dx dz = \frac{\partial(\rho v)}{\partial y} dx dy dz$$

z 方向的净流出量

$$[\rho w + \frac{\partial(\rho w)}{\partial z}]dxdy - (\rho w)dxdy = \frac{\partial(\rho w)}{\partial z}dxdydz$$

那么净流出量

$$= [\frac{\partial(\rho w)}{\partial z} + \frac{\partial(\rho v)}{\partial y} + \frac{\partial(\rho u)}{\partial x}]dxdydz$$

微团的总质量是 $\rho(dxdydz)$, 因此微团内质量增加的时间变化率为 $= \frac{\partial\rho}{\partial t}(dxdydz)$

根据质量守恒定律

$$[\frac{\partial(\rho w)}{\partial z} + \frac{\partial(\rho v)}{\partial y} + \frac{\partial(\rho u)}{\partial x}]dxdydz = -\frac{\partial\rho}{\partial t}(dxdydz)$$

\Rightarrow

$$\frac{\partial\rho}{\partial t} + [\frac{\partial(\rho w)}{\partial z} + \frac{\partial(\rho v)}{\partial y} + \frac{\partial(\rho u)}{\partial x}] = 0$$

\Rightarrow

$$\frac{\partial\rho}{\partial t} + \nabla \cdot (\rho \vec{V}) = 0$$

上述方程式连续性方程的偏微分方程形式. 他是基于空间位置固定的无穷小微团模型.

24.1.12 随流体运动的无穷小微团模型

这个流体微团有固定的质量, 但它的形状和体积会在它向下游运动时变化. 将这个流体微团固定的质量和可变的体积分别用 δm 和 δV 表示.

\Rightarrow

$$\delta m = \rho \delta V$$

他的质量随时间的变化率为0

$$\frac{D(\delta m)}{Dt} = 0$$

\Rightarrow

$$\frac{D(\rho \delta V)}{Dt} = \delta V \frac{D\rho}{Dt} + \rho \frac{D(\delta V)}{Dt} = 0$$

=>

$$\frac{D\rho}{Dt} + \rho \nabla \cdot \vec{V} = 0$$

上式是连续性方程的另一种偏微分方程的形式, 由流体运动模型直接导出的控制方程定义为非守恒型方程.

24.1.13 散度定理

简单来说就是把面积分转为体积分, 也要求数学上的连续性

$$\iint_S (p \mathcal{V}) \cdot d\vec{S} = \iiint_V \nabla \cdot (\rho \vec{V}) d\mathcal{V}$$

24.1.14 对于标量与向量成绩的散度, 有向量恒等式

$$\nabla \cdot (\rho \vec{V}) = (\rho \nabla \cdot \vec{V}) + (\vec{V} \cdot \nabla \rho)$$

也就是说, 一个标量与一个向量成绩的散度等于标量与向量散度的成绩加上向量与标量梯度的点积.

24.1.15 动量方程

$$\vec{F}_x = m \vec{a}_x$$

x方向的动量方程

有两种里会影响真个流体微团, 体积力, 直接作用在流体微团整个体积委员上的力, 而且租用是超距离的, 比如重力, 电场力, 磁场力.

表面力, 直接作用在流体微团的表面, 他们只能由两种原因引起: 1. 由包在流体微团周围的流体所施加的, 作用于微团表面的压力分布; 2. 由于外部流体推拉微团而产生的, 以摩擦的方式作用于表面的切应力和正应力分布.

x轴方向总的表面力 =

$$[p - (p + \frac{\partial p}{\partial x} dx)] dy dz + [(\tau_{xx} + \frac{\partial \tau_{xx}}{\partial x} - \tau_{xx})] dy dz + [(\tau_{yz} + \frac{\partial \tau_{yz}}{\partial y} dy) - \tau_{yz}] dx dz + [(\tau_{zx} + \frac{\partial \tau_{zx}}{\partial z} dz) - \tau_{rz}] dx dy$$

里面含有压力和切应力

作用在单位质量流体微团上的体积力记作 $\{$, 其x方向分量为 $\{_x$. 流体微团的体积为 $dxdydz$, 所以LinkedList还可以当做队列来使用

作用在流体微团上的体积力的x方向风量= $\rho\{_x(dxdydz)$

x方向总的力 F_x

$$F_x = \left(-\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} \right) dxdydz + \rho\{_x dxdydz = ma = \rho \frac{Du}{Dt} dxdydz$$

=>

$$\rho \frac{Dv}{Dt} = -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho\{_y$$

$$\rho \frac{Dw}{Dt} = -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho\{_z$$

上述三个方程也统称为N-V方程. 纳维-斯托克斯方程.

=>

$$\frac{\partial(\rho u)}{\partial t} + \nabla \cdot (\rho u \vec{V}) = \left(-\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} \right) + \rho\{_x$$

$$\frac{\partial(\rho v)}{\partial t} + \nabla \cdot (\rho v \vec{V}) = \left(-\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} \right) + \rho\{_y$$

$$\frac{\partial(\rho w)}{\partial t} + \nabla \cdot (\rho w \vec{V}) = \left(-\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} \right) + \rho\{_z$$

牛顿流体: 流体的切应力与应变的时间变化率, 也就是速度梯度, 是成正比的.

空气动力学中都是牛顿流体.

$$\tau_{xx} = \lambda(\nabla \cdot \vec{V}) + 2\mu \frac{\partial u}{\partial x}$$

$$\tau_{yy} = \lambda(\nabla \cdot \vec{V}) + 2\mu \frac{\partial v}{\partial y}$$

$$\tau_{zz} = \lambda(\nabla \cdot \vec{V}) + 2\mu \frac{\partial w}{\partial z}$$

$$\tau_{xy} = \tau_{yx} = \mu \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right)$$

$$\tau_{xz} = \tau_{zx} = \mu \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right)$$

$$\tau_{yz} = \tau_{zy} = \mu \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right)$$

其中, μ 是分子粘性系数, λ 是第二粘性系数. 斯托克斯提出假设

$$\lambda = -\frac{2}{3}\mu$$

=> 完整的NV守恒方程

$$\frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u^2)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} + \frac{\partial(\rho uw)}{\partial z} = -\frac{\partial p}{\partial x} + \frac{\partial}{\partial t} (\lambda \nabla \cdot \vec{V} + 2\mu \frac{\partial u}{\partial x}) + \frac{\partial}{\partial y} [\mu \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right)] + \frac{\partial}{\partial z} [\mu \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right)] + \rho \{_x$$

$$\begin{aligned} \frac{\partial(\rho v)}{\partial t} + \frac{\partial(\rho uv)}{\partial x} + \frac{\partial(\rho v^2)}{\partial y} + \frac{\partial(\rho vw)}{\partial z} &= -\frac{\partial p}{\partial y} + \frac{\partial}{\partial x} \left[\mu \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \\ \frac{\partial}{\partial y} \left(\lambda \nabla \cdot V + 2\mu \frac{\partial v}{\partial y} \right) + \frac{\partial}{\partial z} \left[\mu \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right) \right] &+ \rho f_y \end{aligned}$$

$$\frac{\partial(\rho w)}{\partial t} + \frac{\partial(\rho uw)}{\partial x} + \frac{\partial(\rho vw)}{\partial y} + \frac{\partial(\rho w^2)}{\partial z} = -\frac{\partial p}{\partial z} + \frac{\partial}{\partial x} \left[\mu \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \right] + \frac{\partial}{\partial y} \left[\mu \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right) \right] + \frac{\partial}{\partial z} \left(\lambda \nabla \cdot V + 2\mu \frac{\partial w}{\partial z} \right)$$

24.1.16 能量方程

流体微团内能量的变化率=流入微团内的净热流量+体积力和表面力对微团做功的功率

$$\mathbf{A} = \mathbf{B} + \mathbf{C}$$

\mathbf{C} 为, 对物体做功的功率等于这个力乘以速度在此力作用方向的分量.

$$\rho \{ \cdot \vec{V} (dxdydz)$$

压力在x方向做功的功率为

$$\left[up - \left(up + \frac{\partial(up)}{\partial x} dx \right) \right] dy dz = -\frac{\partial(up)}{\partial x} dx dy dz$$

切应力在x方向上做功的功率是

$$\left[\left(u\tau_{yx} + \frac{\partial(u\tau_{yx})}{\partial y} dy \right) - u\tau_{yx} \right] dx dz = \frac{\partial(u\tau_{yx})}{\partial y} dx dy dz$$

所有表面力x方向的做功的和

$$\left[-\frac{\partial(up)}{\partial x} + \frac{\partial(u\tau_{xx})}{\partial x} + \frac{\partial(u\tau_{yx})}{\partial y} + \frac{\partial(u\tau_{zx})}{\partial z} \right] dx dy dz$$

xyz方向上的做功和

$$C = - \left[\left(\frac{\partial(up)}{\partial x} + \frac{\partial(vp)}{\partial y} + \frac{\partial(wp)}{\partial z} \right) + \frac{\partial(u\tau_{xx})}{\partial x} + \frac{\partial(u\tau_{yx})}{\partial y} + \frac{\partial(u\tau_{zx})}{\partial z} + \frac{\partial(v\tau_{xy})}{\partial x} + \right. \\ \left. \frac{\partial(v\tau_{ry})}{\partial y} + \frac{\partial(v\tau_{zy})}{\partial z} + \frac{\partial(w\tau_{zz})}{\partial x} + \frac{\partial(w\tau_{yz})}{\partial y} + \frac{\partial(w\tau_z)}{\partial z} \right] dx dy dz + \rho f \cdot V dx dy dz$$

\dot{q} 定义为单位质量的体积加热率.

微团的体积加热(吸收或释放的辐射热) = $\dot{q}\rho dx dy dz$

$$\text{热传导对流体微团的加热} = - \left(\frac{\partial \dot{q}_x}{\partial x} + \frac{\partial \dot{q}_y}{\partial y} + \frac{\partial \dot{q}_z}{\partial z} \right) dx dy dz$$

$$B = \left[\rho \dot{q} - \left(\frac{\partial \dot{q}_x}{\partial x} + \frac{\partial \dot{q}_y}{\partial y} + \frac{\partial \dot{q}_z}{\partial z} \right) \right] dx dy dz$$

根据傅里叶热传导定律, 热传导产生的热流与当地的问题梯度成正比

$$\dot{q}_x = -k \frac{\partial T}{\partial x} \quad \dot{q}_y = -k \frac{\partial T}{\partial y} \quad \dot{q}_z = -k \frac{\partial T}{\partial z}$$

k 为热导率

=>

$$B = \left[\rho \dot{q} + \frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) \right] dx dy dz$$

流体微团内能量= 由于分子随机运动而产生的(单位质量)内能e + 流体微团平动是具有的动能. 单位质量的动能为 $V^2/2$

单位质量的总能量变化的时间变化率由物质导数给出.

$$A = \rho \frac{D}{Dt} \left(e + \frac{V^2}{2} \right) dx dy dz$$

=> 总公式

$$\begin{aligned} \rho \frac{D}{Dt} \left(e + \frac{V^2}{2} \right) = & \rho \dot{q} + \frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) - \\ & \frac{\partial (up)}{\partial x} - \frac{\partial (vp)}{\partial y} - \frac{\partial (wp)}{\partial z} + \frac{\partial (u\tau_{xx})}{\partial x} + \frac{\partial (u\tau_{yx})}{\partial y} + \frac{\partial (u\tau_{zx})}{\partial z} + \\ & \frac{\partial (v\tau_{xy})}{\partial x} + \frac{\partial (v\tau_{yy})}{\partial y} + \frac{\partial (v\tau_{zy})}{\partial z} + \frac{\partial (w\tau_{xz})}{\partial x} + \frac{\partial (w\tau_{yz})}{\partial y} + \frac{\partial (w\tau_{zz})}{\partial z} + \rho f \cdot V \end{aligned}$$

=> 将方程左边只包含内能

$$\begin{aligned} \rho \frac{De}{Dt} = & \rho \dot{q} + \frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) - p \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \right) + \\ & \tau_{xx} \frac{\partial u}{\partial x} + \tau_{yx} \frac{\partial u}{\partial y} + \tau_{zz} \frac{\partial u}{\partial z} + \tau_{xy} \frac{\partial v}{\partial x} + \tau_{yy} \frac{\partial v}{\partial y} + \tau_{zy} \frac{\partial v}{\partial z} + \tau_{xz} \frac{\partial w}{\partial x} + \tau_{yz} \frac{\partial w}{\partial y} + \tau_{zz} \frac{\partial w}{\partial z} \end{aligned}$$

=> 利用 $\tau_{xy} == \tau_{yx} \dots$

$$\begin{aligned} \rho \frac{De}{Dt} = & \rho \dot{q} + \frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) - p \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \right) + \\ & \lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \right)^2 + \mu \left[2 \left(\frac{\partial u}{\partial x} \right)^2 + 2 \left(\frac{\partial v}{\partial y} \right)^2 + 2 \left(\frac{\partial w}{\partial z} \right)^2 + \right. \\ & \left. \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right)^2 + \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right)^2 + \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right)^2 \right] \end{aligned}$$

=> 内能表示的守恒型能量方程

$$\begin{aligned} \frac{\partial(\rho e)}{\partial t} + \nabla \cdot (\rho e V) = & \rho \dot{q} + \frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) - \\ & p \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \right) + \lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \right)^2 + \\ & \mu \left[2 \left(\frac{\partial u}{\partial x} \right)^2 + 2 \left(\frac{\partial v}{\partial y} \right)^2 + 2 \left(\frac{\partial w}{\partial z} \right)^2 + \right. \\ & \left. \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right)^2 + \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right)^2 + \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right)^2 \right] \end{aligned}$$

=> 总能量表示的守恒型方程

$$\begin{aligned}
& \frac{\partial}{\partial t} \left[\rho \left(e + \frac{V^2}{2} \right) \right] + \nabla \cdot \left[\rho \left(e + \frac{V^2}{2} \right) V \right] \\
&= \rho \dot{q} + \frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) - \frac{\partial (up)}{\partial x} - \frac{\partial (vp)}{\partial y} - \frac{\partial (wp)}{\partial z} + \\
& \quad \frac{\partial (u\tau_{xx})}{\partial x} + \frac{\partial (u\tau_{yx})}{\partial y} + \frac{\partial (u\tau_{zx})}{\partial z} + \frac{\partial (v\tau_{xy})}{\partial x} + \frac{\partial (v\tau_{yy})}{\partial y} + \frac{\partial (v\tau_{zy})}{\partial z} + \\
& \quad \frac{\partial (w\tau_{xz})}{\partial x} + \frac{\partial (w\tau_{yz})}{\partial y} + \frac{\partial (w\tau_{zz})}{\partial z} + \rho f \cdot V
\end{aligned}$$

将非守恒型改成守恒型, 只要更改方程左边即可

24.1.17 粘性流动的NS方程

- a. 连续性方程
- 1. 非守恒形式

$$\frac{D\rho}{Dt} + \rho \nabla \cdot V = 0$$

- 2. 守恒形式

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho V) = 0$$

- b. 动量方程

- 1. 非守恒形式

$$\begin{aligned}
x \text{ 方向} \quad & \rho \frac{Du}{Dt} = -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \\
y \text{ 方向} \quad & \rho \frac{Dv}{Dt} = -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho f_y \\
z \text{ 方向} \quad & \rho \frac{Dw}{Dt} = -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho f_z
\end{aligned}$$

- 2. 守恒形式

$$\begin{aligned}
x \text{ 方向} \quad & \frac{\partial(\rho u)}{\partial t} + \nabla \cdot (\rho u V) = -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \\
y \text{ 方向} \quad & \frac{\partial(\rho v)}{\partial t} + \nabla \cdot (\rho v V) = -\frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho f_y \\
z \text{ 方向} \quad & \frac{\partial(\rho w)}{\partial t} + \nabla \cdot (\rho w V) = -\frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho f_z
\end{aligned}$$

- c. 能量方程

- 1. 非守恒形式

$$\rho \frac{D}{Dt} \left(e + \frac{V^2}{2} \right) = \rho \dot{q} + \frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) - \frac{\partial (up)}{\partial x} -$$

$$\frac{\partial (vp)}{\partial y} - \frac{\partial (wp)}{\partial z} + \frac{\partial (u\tau_{xx})}{\partial x} + \frac{\partial (u\tau_{yx})}{\partial y} + \frac{\partial (u\tau_{zx})}{\partial z} +$$

$$\frac{\partial (v\tau_{xy})}{\partial x} + \frac{\partial (v\tau_{yy})}{\partial y} + \frac{\partial (u\tau_{zy})}{\partial z} + \frac{\partial (w\tau_{xz})}{\partial x} + \frac{\partial (w\tau_{yz})}{\partial y} +$$

$$\frac{\partial (w\tau_z)}{\partial z} + \rho f \cdot V$$

2. 守恒形式

$$\begin{aligned} & \frac{\partial}{\partial t} \left[\rho \left(e + \frac{V^2}{2} \right) \right] + \nabla \cdot \left[\rho \left(e + \frac{V^2}{2} \right) V \right] \\ &= \rho \dot{q} + \frac{\partial}{\partial x} \left(k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left(k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) - \frac{\partial (up)}{\partial x} - \frac{\partial (vp)}{\partial y} - \frac{\partial (w\tau_z)}{\partial z} \\ & \quad \frac{\partial (u\tau_{xx})}{\partial x} + \frac{\partial (u\tau_{yx})}{\partial y} + \frac{\partial (u\tau_{zx})}{\partial z} + \frac{\partial (v\tau_{xy})}{\partial x} + \frac{\partial (v\tau_{yy})}{\partial y} + \frac{\partial (u\tau_{zy})}{\partial z} + \\ & \frac{\partial (w\tau_{xz})}{\partial x} + \frac{\partial (w\tau_{yz})}{\partial y} + \frac{\partial (w\tau_{zz})}{\partial z} + \rho f \cdot V \end{aligned}$$

24.1.18 无粘性流动的NS方程

忽略了耗散, 粘性输运, 质量扩散以及热传导的流动. 如果我们采用2.8.1小节列出的方程, 并且简单地去掉其中所有包含摩擦和热传导的项, 就得到了无粘流动的方程.

1. 连续性方程

非守恒形式

$$\frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{V} = 0$$

守恒形式

$$\frac{\partial}{\partial t} + \nabla \cdot (\rho \mathbf{V}) = 0$$

2. 动量方程

非守恒形式

x方向, y方向, z方向

$$\begin{aligned}\rho \frac{Du}{Dt} &= -\frac{\partial p}{\partial x} + \rho f_x \\ \rho \frac{Dv}{Dt} &= -\frac{\partial p}{\partial y} + \rho f_y \\ \rho \frac{Dw}{Dt} &= -\frac{\partial p}{\partial z} + \rho f_z\end{aligned}$$

守恒形式

x方向, y方向, z方向

$$\begin{aligned}\frac{\partial(\rho u)}{\partial t} + \nabla \cdot (\rho u V) &= -\frac{\partial p}{\partial x} + \rho f_x \\ \frac{\partial(\rho v)}{\partial t} + \nabla \cdot (\rho v V) &= -\frac{\partial p}{\partial y} + \rho f_y \\ \frac{\partial(\rho w)}{\partial t} + \nabla \cdot (\rho w V) &= -\frac{\partial p}{\partial z} + \rho f_z\end{aligned}$$

3. 能量方程

非守恒形式

$$\rho \frac{D}{Dt} \left(e + \frac{V^2}{2} \right) = \rho \dot{q} - \frac{\partial (up)}{\partial x} - \frac{\partial (vp)}{\partial y} - \frac{\partial (wp)}{\partial z} + \rho f \cdot V$$

守恒形式

$$\frac{\partial}{\partial t} \left[\rho \left(e + \frac{V^2}{2} \right) \right] + \nabla \cdot \left[\rho \left(e + \frac{V^2}{2} \right) V \right] = \rho \dot{q} - \frac{\partial (up)}{\partial x} - \frac{\partial (vp)}{\partial y} - \frac{\partial (wp)}{\partial z} + \rho f \cdot V$$

24.1.19 无粘性流动物理边界条件

$$\vec{V} \cdot \vec{n} = 0$$

在物面

24.1.20 强守恒和弱守恒

所有的東西都寫進了導數裏面，稱為強守恒，反之，稱為弱守恒

24.1.21 特征线

偏微分方程中, $|A| = 0$ 可以找到特征线, $\frac{\partial u}{\partial x}$, 的值时不确定的.

然后可以得到一个类似, 2次方程, 求解其方程, 就可以得到求解方式

$D > 0$, 给出的方程组称为双曲型方程组

$D = 0$, 给出的方程组称为抛物型方程组

$D < 0$, 特征线是虚的, 给出的方程称为椭圆型方程组

24.1.22 特征值

$$BX = A$$

24.1.23 双曲线 & 抛物线

可以通过推进法求解.

24.1.24 椭圆

陪审团问题. 解依赖于所有边界.

1. 在边界上指定未知函数u和v, 这种边界条件称为Dirichlet条件
2. 在边界上指定未知函数的导数, 例如 $\frac{\partial u}{\partial x}$ 这种边界条件称为Nenmann条件
3. Dirichlet条件和Neumann条件的混合条件

应用条件

1. 定长亚声速无粘流动
2. 不可压无粘流动

24.1.25 定义适定性

如果一个偏微分方程的解存在并且是惟一的, 同时, 解连续地依赖于初始条件和边界条件, 那么这个问题就是适定性的

24.1.26 偏微分形式和积分形式的求解

偏微分方程的离散化, 有限差分法.

积分形式方程的离散化称为, 有限体积法.

24.1.27 CFD的求解方法分类

1. 有限元
2. 有限差分法
3. 有限体积法

24.1.28 前向差分

$$\left(\frac{\partial u}{\partial x} \right)_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + O(\Delta x)$$

24.1.29 后向差分

$$\left(\frac{\partial u}{\partial x} \right)_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x} + O(\Delta x)$$

24.1.30 二阶中心差分

$$\left(\frac{\partial u}{\partial x} \right)_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + O(\Delta x)^2$$

$$\left(\frac{\partial^2 u}{\partial x^2} \right)_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + O(\Delta x)^2$$

$$\left(\frac{\partial^2 u}{\partial y^2} \right)_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} + O(\Delta y)^2$$

混合二阶

$$\left(\frac{\partial^2 u}{\partial x \partial y} \right)_{i,j} = \frac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{4\Delta x \Delta y} + O[(\Delta x)^2, (\Delta y)^2]$$

24.1.31 边界处理

1. 使用假设边界的下面的值就等于边界的值来回避这个问题
2. 使用多项式近似的方法来对边界值进行求解

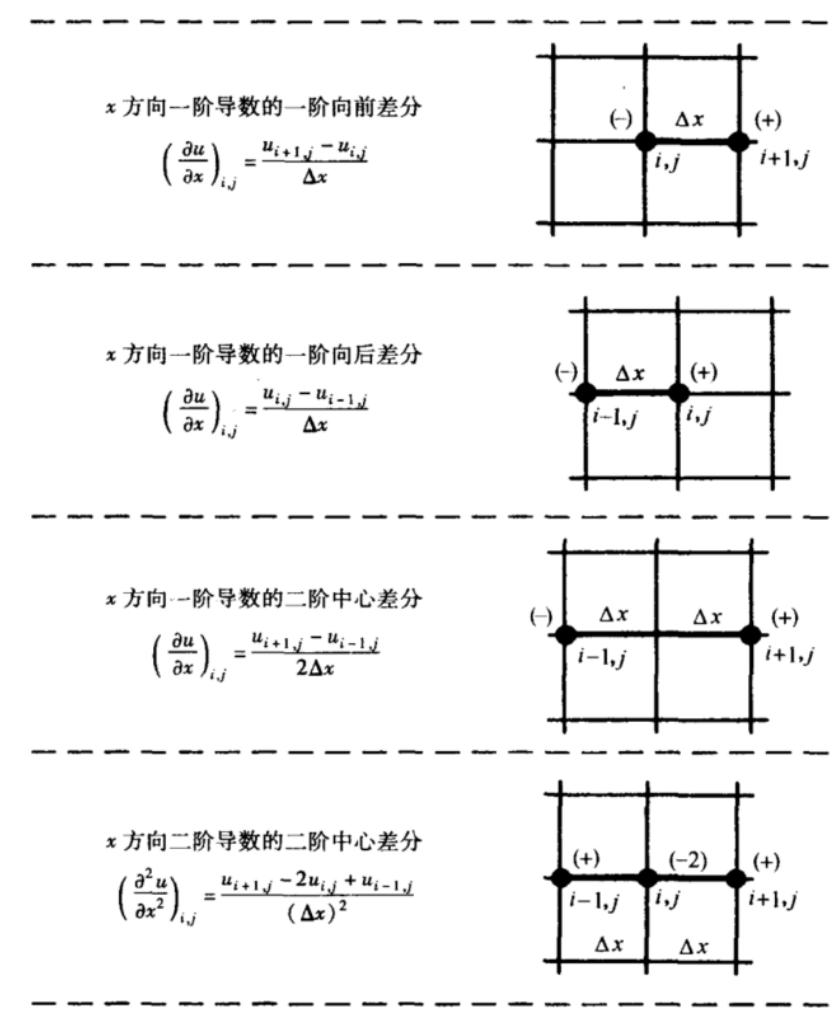


图 24-1 youxianchafentujie

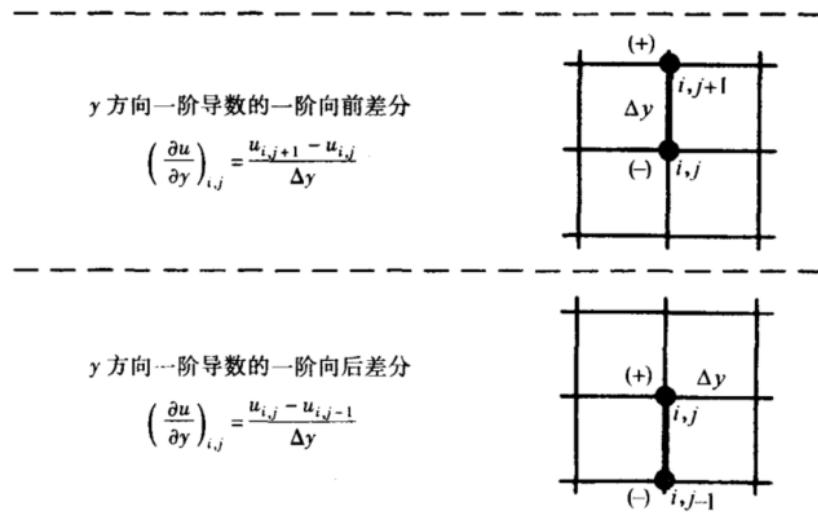


图 4-3 有限差分表达式及相应的有限差分模板

图 24-2 youxianchafentujie2

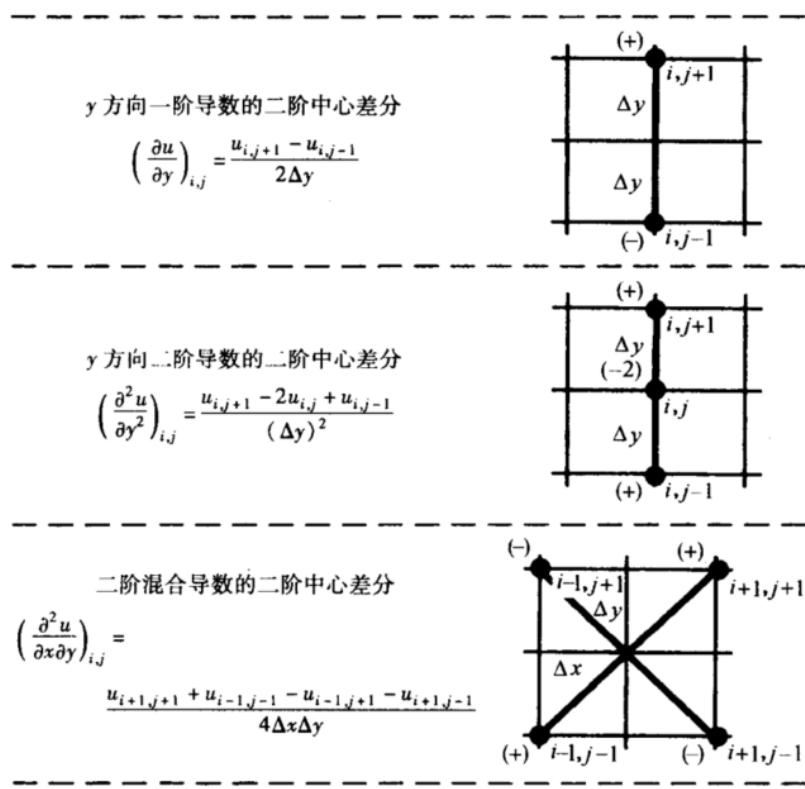


图 4-3 有限差分表达式及相应的有限差分模板(续)

图 24-3 youxianchafentujie3

24.1.32 显式方法

显式方法中每一个差分方程只包含一个未知数, 从而这个未知数可以用直接计算的方式显式地求解.

优点: 方法的建立及编程相对简单

缺点: 更具上面的例子, 对确定的 $\Delta x, \Delta t$ 要尽量小, 否则容易出现不稳定状态, 导致了计算时间耗时较长.

24.1.33 隐式方法

对于排列在同一时间层所有网格点上的未知量, 必须将它们联立起来同时求解, 才能求出这些未知量. – 采用追赶法, 托马斯算法.

优点: 耗时更小, Δt 显示的更加稳定

缺点: 方法的建立和编程更复杂.

24.1.34 波数

下标m的含义, 它就等于给定区间包含的波形的个数.

24.1.35 稳定解判定

$$\left| \frac{\varepsilon_i^{n+1}}{\varepsilon_i^n} \right| = |e^{\alpha \Delta t}| = \left| 1 - \frac{4\alpha \Delta t}{(\Delta x)^2} \sin^2 \frac{k_m \Delta x}{2} \right| \leq 1$$

分析方法, 冯诺伊曼稳定性方法.

柯朗-弗里德里奇-列为条件, 这个天剑对于双曲型方程来说是一个重要的稳定性准则. 要保证稳定性, 数值解的依赖区域必须全部包含解析解的依赖区域.

24.1.36 网格变换

简单的来说就是建立一个联系, 将物理坐标和计算坐标建立联系

$$\frac{\partial}{\partial x} = \left(\frac{\partial}{\partial \xi} \right) \left(\frac{\partial \xi}{\partial x} \right) + \left(\frac{\partial}{\partial \eta} \right) \left(\frac{\partial \eta}{\partial x} \right)$$

$$\frac{\partial}{\partial y} = \left(\frac{\partial}{\partial \xi} \right) \left(\frac{\partial \xi}{\partial y} \right) + \left(\frac{\partial}{\partial \eta} \right) \left(\frac{\partial \eta}{\partial y} \right)$$

$$\frac{\partial}{\partial t} = \left(\frac{\partial}{\partial \xi} \right) \left(\frac{\partial \xi}{\partial t} \right) + \left(\frac{\partial}{\partial \eta} \right) \left(\frac{\partial \eta}{\partial t} \right) + \left(\frac{\partial}{\partial \tau} \right) \left(\frac{d\tau}{dt} \right)$$

$$\begin{aligned} \frac{\partial^2}{\partial x^2} &= \left(\frac{\partial}{\partial \xi} \right) \left(\frac{\partial^2 \xi}{\partial x^2} \right) + \left(\frac{\partial}{\partial \eta} \right) \left(\frac{\partial^2 \eta}{\partial x^2} \right) + \left(\frac{\partial^2}{\partial \xi^2} \right) \left(\frac{\partial \xi}{\partial x} \right)^2 + \left(\frac{\partial^2}{\partial \eta^2} \right) \left(\frac{\partial \eta}{\partial x} \right)^2 + \\ &\quad 2 \left(\frac{\partial^2}{\partial \eta \partial \xi} \right) \left(\frac{\partial \eta}{\partial x} \right) \left(\frac{\partial \xi}{\partial x} \right) \end{aligned}$$

$$\begin{aligned} \frac{\partial^2}{\partial y^2} &= \left(\frac{\partial}{\partial \xi} \right) \left(\frac{\partial^2 \xi}{\partial y^2} \right) + \left(\frac{\partial}{\partial \eta} \right) \left(\frac{\partial^2 \eta}{\partial y^2} \right) + \left(\frac{\partial^2}{\partial \xi^2} \right) \left(\frac{\partial \xi}{\partial y} \right)^2 + \\ &\quad \left(\frac{\partial^2}{\partial \eta^2} \right) \left(\frac{\partial \eta}{\partial y} \right)^2 + 2 \left(\frac{\partial^2}{\partial \eta \partial \xi} \right) \left(\frac{\partial \eta}{\partial y} \right) \left(\frac{\partial \xi}{\partial y} \right) \end{aligned}$$

$$\begin{aligned} \frac{\partial^2}{\partial x \partial y} &= \left(\frac{\partial}{\partial \xi} \right) \left(\frac{\partial^2 \xi}{\partial x \partial y} \right) + \left(\frac{\partial}{\partial \eta} \right) \left(\frac{\partial^2 \eta}{\partial x \partial y} \right) + \left(\frac{\partial^2}{\partial \xi^2} \right) \left(\frac{\partial \xi}{\partial x} \right) \left(\frac{\partial \xi}{\partial y} \right) + \\ &\quad \left(\frac{\partial^2}{\partial \eta^2} \right) \left(\frac{\partial \eta}{\partial x} \right) \left(\frac{\partial \eta}{\partial y} \right) + \left(\frac{\partial^2}{\partial \xi \partial \eta} \right) \left[\left(\frac{\partial \eta}{\partial x} \right) \left(\frac{\partial \xi}{\partial y} \right) + \left(\frac{\partial \xi}{\partial x} \right) \left(\frac{\partial \eta}{\partial y} \right) \right] \end{aligned}$$

因为比较复杂, 使用逆度量来减少复杂度

$$\frac{\partial}{\partial x} = \frac{1}{J} \left[\left(\frac{\partial}{\partial \xi} \right) \left(\frac{\partial y}{\partial \eta} \right) - \left(\frac{\partial}{\partial \eta} \right) \left(\frac{\partial y}{\partial \xi} \right) \right]$$

$$\frac{\partial}{\partial y} = \frac{1}{J} \left[\left(\frac{\partial}{\partial \eta} \right) \left(\frac{\partial x}{\partial \xi} \right) - \left(\frac{\partial}{\partial \xi} \right) \left(\frac{\partial x}{\partial \eta} \right) \right]$$

$$\begin{aligned} \frac{\partial \xi}{\partial x} &= \frac{1}{J} \frac{\partial y}{\partial \eta} \\ \frac{\partial \eta}{\partial x} &= -\frac{1}{J} \frac{\partial y}{\partial \xi} \\ \frac{\partial \xi}{\partial y} &= -\frac{1}{J} \frac{\partial x}{\partial \eta} \\ \frac{\partial \eta}{\partial y} &= \frac{1}{J} \frac{\partial x}{\partial \xi} \end{aligned}$$

24.1.37 自适应网格

1. 当网格数量固定时, 可以提高计算精度
2. 给定精度时, 可以用较少的网格点来达到这一精度.
笛卡尔积网格, 网格靠近物体的边界直接向物体拟合.

24.1.38 显示有限差分方法

1. 拉克斯-温德罗夫Lax-Wendroff方法

2. 麦考马克(MacCormack)方法

最容易理解和编程的方法之一.

应用于非定常纳维-斯托克斯方程的求解.

3. 松弛法

如果截断误差的主项是偶数阶导数, 数值解将主要表现出耗散行为; 如果主项是奇数阶导数, 数值解将主要表现出色散行为.

人工粘性加入可以得到一个稳定的解.

$$\begin{aligned} \overline{S_{i,j}^{t+\Delta t}} &= C_x \frac{\left| \overline{p_{i+1,j}^{t+\Delta t}} - 2\overline{p_{i,j}^{t+\Delta t}} + \overline{p_{i-1,j}^{t+\Delta t}} \right|}{\overline{p_{i+1,j}^{t+\Delta t}} + 2\overline{p_{i,j}^{t+\Delta t}} + \overline{p_{i-1,j}^{t+\Delta t}}} \left(\overline{U_{i+1,j}^{t+\Delta t}} - 2\overline{U_{i,j}^{t+\Delta t}} + \overline{U_{i-1,j}^{t+\Delta t}} \right) + \\ &C_y \frac{\left| \overline{\bar{p}_{i,j+1}^{t+\Delta t}} - 2\overline{\bar{p}_{i,j}^{t+\Delta t}} + \overline{\bar{p}_{i,j-1}^{t+\Delta t}} \right|}{\overline{p_{i,j+1}^{t+\Delta t}} + 2\overline{p_{i,j}^{t+\Delta t}} + \overline{p_{i,j-1}^{t+\Delta t}}} \left(\overline{U_{i,j+1}^{t+\Delta t}} - 2\overline{U_{i,j}^{t+\Delta t}} + \overline{U_{i,j-1}^{t+\Delta t}} \right) \end{aligned}$$

24.1.39 交替方向隐式(ADI)方法

简单的说, 将不能使用, 托马斯方法的矩阵变成可以使用托马斯方法??

24.1.40 压力修正法

不可压缩纳维-斯托克斯方程

连续性方程	$\nabla \cdot V = 0$
x 方向动量方程	$\rho \frac{Du}{Dt} = -\frac{\partial p}{\partial x} + \mu \nabla^2 u + \rho f_x$
y 方向动量方程	$\rho \frac{Dv}{Dt} = -\frac{\partial p}{\partial y} + \mu \nabla^2 v + \rho f_y$
z 方向动量方程	$\rho \frac{Dw}{Dt} = -\frac{\partial p}{\partial z} + \mu \nabla^2 w + \rho f_z$

迎风差分. 简答说就是在不同位置就算u, 在不同位置计算v.

24.1.41 数值方法: SIMPLE算法

Semi-implicit method for pressure-linked equations(压力耦合方程的半隐式算法)

24.1.42 马赫数

马赫数 (Ma)

这是流体力学中表征流体可压缩程度的一个重要的无量纲参数，记为Ma，定义为流场中某点的速度v 同该点的当地声速c 之比，它是以奥地利科学家E.马赫的姓氏命名的。

$$Ma = \frac{v}{c}$$

由定义可知，马赫数是表示声速倍数的数，一马赫即一倍音速：马赫数小于1者为亚音速，近乎等于1为跨声速，大于1为超声速。

24.2 计算流体力学视频

https://www.bilibili.com/video/BV1vE411W7kV?from=search&seid=17354358762961406319&spm_id_from=333.337.0.0

25

申论

► ...

25.1 知识点和方法论

25.1.1 xi总书记名言

绿水青山就是金山银山

25.1.2 2022年国考

时间长河奔腾不息，当今中国，正处在由大向强的“关键一跃”，夺目的成就令人骄傲。但是在这继往开来的新时代，仍然面临许多挑战，国际封锁打压仍在持续，国内乡村振兴方才起步、经济下行压力渐增，如何破局？我认为要做好融合的大文章。

共融造就共荣。人类文明演进万年，每一次生产要素的流动与融合都极大地推进了时代的进化。火与食物的融合，开发了人类的大脑，使智慧迸发；铁与土地的融合，促进了粮食的生产，使人口增加；煤炭电力与机器的融合，推动了创新，使科技爆炸。融合造就了繁荣的历史，我们应毫无疑问地充分肯定融合的价值，用好融合的智慧开辟更为繁荣的未来。正是在这一智慧的指引下，中国处处正在发生着融合更新。从地区上看，大湾区融合激发了粤港澳三地的活力；从地域上看，城乡互动造就了同频共振的欣欣向荣；从产业上看，文旅与交通的融合缔造出无数的网红景点。

融合需要融和。融合不是相加而是相融，不是简单的拼凑嫁接，而是渗透交织、融化和谐。倘若只是将要素简单拼凑，并不会产生良性化学反应，反而可能使各要素相

互攻击、彼此掣肘，可谓适得其反、得不偿失。融和的关键在于统筹，统筹意味着分工，能使得各要素各就其位、各展所长，比如“大蜀道文化联盟”的成立吸引了30余个市县区，通过共同开发相互配合，通过各展所长错位竞争，实现了共荣共赢;统筹也意味着部署，能保证各要素各美其美、相互浸润，比如F市第三中学大胆创新探索，构建“德智体美劳”五大类课程群，建立课程有机融合系统，培养出一批批德智体美劳全面发展的优质青年。

融合更需要融活。融合不是目的，融活才是目的，任何要素的交融理应产生更有活力的结果，惟有如此，融合才是有价值的。如何才能保证融活?我认为需要建立机制。通过机制订立规矩、搭建渠道，使各要素有序流动、按需交融，好比人之血管维系人之生机。山南乡农村公路养护因缺少运作机制，政府部门虽然共管一事但各自为政，养护效率极为低下，养护结果问题百出。与之相反，Z城不靠海、不沿边、无矿产、缺交通，原生的发展要素极度匮乏，却能依靠当地政府搭建起的城乡要素融合机制，引导工商资本下乡“输血”，引进工业资本反哺农业，打造出三产交贯融合全产业链体系，让农民增加收入，让土地焕发活力。两相对比答案自明，必须建立要素流动机制保障融活。

回望历史，时代大吵由浪花卷懂;望眼前路，伟业大道以石子路铺就。让我们始终站在时代潮头，以融合与融活的只会绘就新时代的精彩篇章。