八股文背诵合集

The Sea and Sheng

2021 年 8 月 26 日

目录

1 Spring

1.0.1 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.

1.0.2 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

1.0.3 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.

1.0.4 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 calib 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

1.0.5 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, 骚婊得到所有的 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

1.0.6 Autowired

使用 byType 和 byName 去寻找需要进行注入的对象

1.0.7 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.conext 包构成了 Spring 框架 Ioc 的基础. 主要使用文件 applicationContext.xml 来进行配置.

1.0.8 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 Sping 注入到 A 中.

1.0.9 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

1.0.10 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.

1.0.11 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

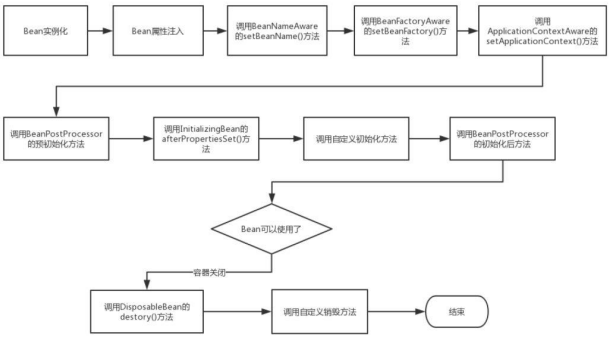

图 1: beanlive

(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() 方法。

(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。

1.0.12 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) 处理器映射器更具请求 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.

1.0.13 第三个三级标题

2 java 基础

2.0.1 如何防止 SQL 注入

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

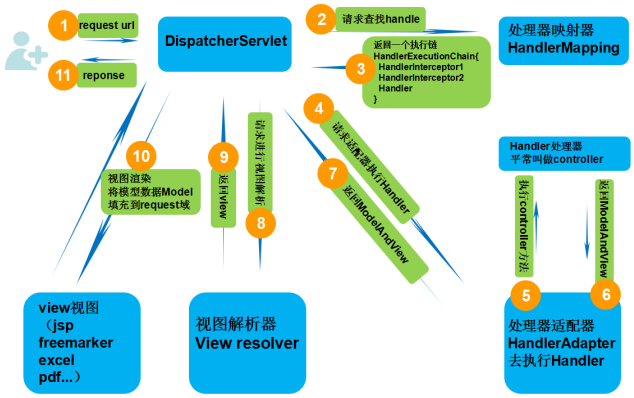

图 2: SpringModelAndView

2.0.2 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

2.0.3 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, 主要关于两个指正, e 指正指向当前节点, 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.

2.0.4 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

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, 将每组分给不同的额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.

2.0.5 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

2.0.6 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

2.0.7 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

2.0.8 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

2.0.9 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

2.0.10 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: 能够对分布在多台季启忠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 中的数据也是在内容, 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()

2.0.11 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 INFOMATION_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

2.0.12 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

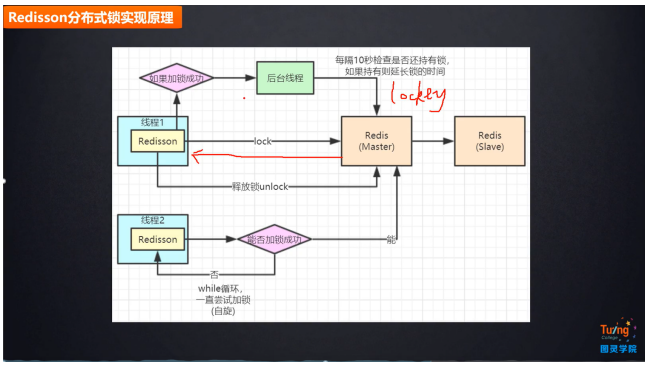

图 3: redission

2. 如果是通过共享内存来进行通信

2.0.13 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, 而安全失败的迭代去永远不会抛出这样的异常.

2.0.14 异常

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

2.0.15 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）

2.0.16 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) threadFacory: 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 默认是丢出异常.

2.0.17 sychronized 和 ReentrantLock 的区别

1. sychronized 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. sychronized 的底层是 jvm 层面的锁, ReentrantLock 是 API 层面的锁

4. sychronized 是非公平锁, ReentrantLock 可以选择公平锁或非公平锁

5. sychronized 锁的是对象, 锁信息保存在对象头中, ReentrantLock 锁的是线程

6. sychronized 底层有一个锁升级的过程

2.0.18 线程池中使用的 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，

2.0.19 Executors 工厂类实现线程池

通过创建不同的 ThreadPoolExecutor 参数.

(1) FixedThreadPool 定长, corePoolSize == maxmumPoolSize

(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 maxmumPoolSize 为无限大, 容易创建大量线程, 从而耗尽系统资源.

2.0.20 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.

2.0.21 8 种基本数据类型

表 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)

2.0.22 Comparable & Comparator 区别

Comparable 是接口能力赋予

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

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

2.0.23 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.

2.0.24 java 深拷贝和浅拷贝

实现了 Coneable 接口实现深拷贝.

2.0.25 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.

2.0.26 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.

2.0.27 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.

2.0.28 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.

2.0.29 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.

2.0.30 内部类

(1) 静态内部类访问外部变量必须是静态的.

(2)

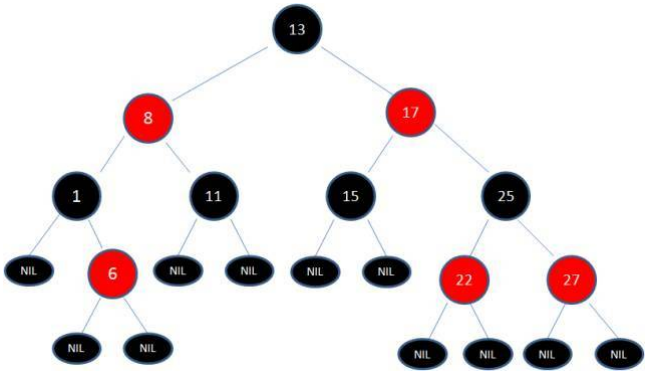

图 4: redblack

2.0.31 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.

2.0.32 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.

2.0.33 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

2.0.34 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.

2.0.35 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, 不会随方法的结束二小时. 方法中的局部变量使用 final 修饰后, 放在堆中, 而不是栈中.

2.0.36 Array 和 ArrayList 的区别

1. Array 大小固定, ArrayList 大小是动态变化的.

2.0.37 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.

2.0.38 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.

2.0.39 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 defalut, 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类覆盖。
12         */
13         default public void defaultMethod(String str){
14                 System.out.println("InterfaceJDK8:" +
                       str);
15         }
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, 也需要提供方法的
               现。
19         */
20         public static void staticMethod(String str){
21                 System.out.println("InterfaceJDK8:" +
                       str);
22         }
23
24 }
```

2.0.40 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.

2.0.41 == 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.

2.0.42 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.

2.0.43 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.

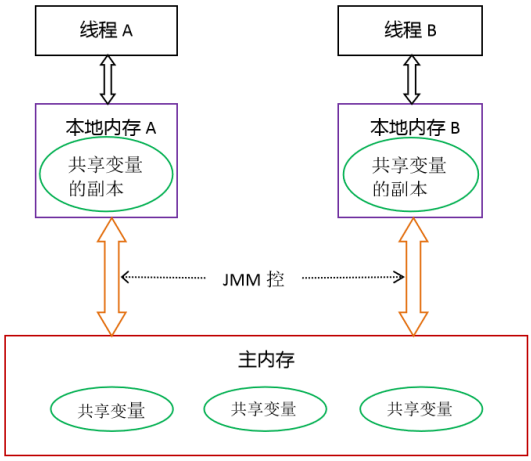

图 5: JMM

2.0.44 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.

2.0.45 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

2.0.46 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.

2.0.47 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

2.0.48 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.

2.0.49 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, final
                Condition consumerCondition) {
16                  this.lock = lock;
17                  this.producerCondition =
                        producerCondition;
18                  this.consumerCondition =
                        consumerCondition;
19          }
20
21          /**
22          * 生产消息
23          * */
24          public void set() {
25                  /** 获取锁*/
26                  lock.lock();
27                  try {
28                          if (count <= maximum) {
29                                  /** 生产一个消息*/
30                                  System.out.println("生
                                        产者 线程" + Thread
                                        .currentThread().
                                        getName() + "生产了
                                        一个消息, 当前有" +
                                        (++count) + "个消
                                        息");
31                                  /** 唤醒等待的消费者*/
32                                  consumerCondition.
                                        signal();
33                          } else {
34                                  try {
35                                          /**
36                                          * 如果当前消息
```

```
                                    大于
                                    maximum信息
                                    最大数
37                                  * 生产者进入睡
                                    眠/等待状态
38                                  * */
39                                  producerCondition
                                    .await();
40                                  System.out.
                                    println("生
                                    产者 线程"
                                    + Thread.
                                    currentThread
                                    ().getName
                                    () + "进入
                                    睡眠");
41                              } catch (
                                    InterruptedException
                                     e) {
42                                  e.
                                    printStackTrace
                                    ();
43                              }
44                      }
45              } finally {
46                      /** 释放锁*/
47                      lock.unlock();
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,
                        consumerCondition);
131                 /** 建几个生产线程*/
132                 new Thread(new Producer(message)).
                        start();
133                 new Thread(new Producer(message)).
                        start();
134                 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 }
```

2.0.50 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

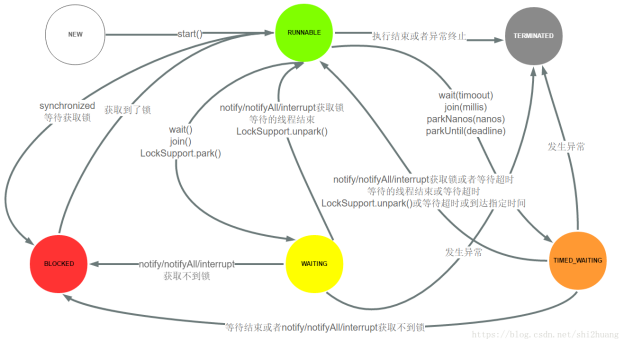

图 6: javaThreadState

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.

2.0.51 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

2.0.52 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 修改对象投里的锁标志位, 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

2.0.53 如何使用 synchronized

1. 普通同步方法

3 JVM

3.0.1 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

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, 就是正常的对象, 没有被找到的对象可以被回收.

3.0.2 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

3.0.3 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

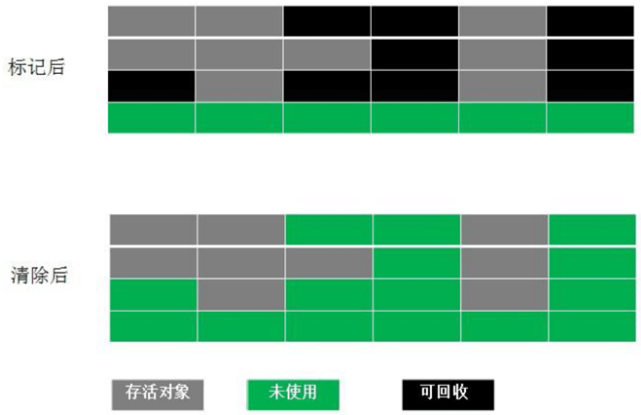

图 7: signremove

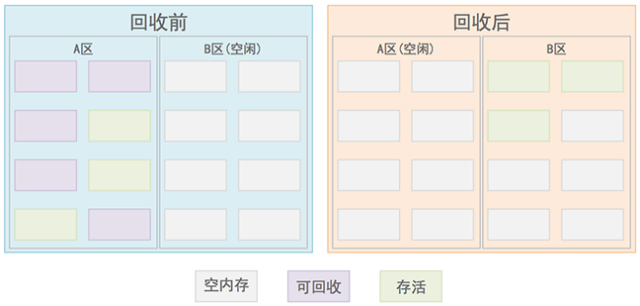

图 8: jvmcopy

3. 复制算法: 分别 8:1 的 Eden 去和 survivor 区

4. 分代收集算法-重点

   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.

3.0.4 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 engin(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: 根据给定的额全限定类名 (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: 首先通过编译器吧 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) 的方发区内, 而字节码文件知识 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去内, 然后在堆区创建一个 java.lang.Class 对象, 用来封装类在方区内的数据结构.

3.0.5 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.

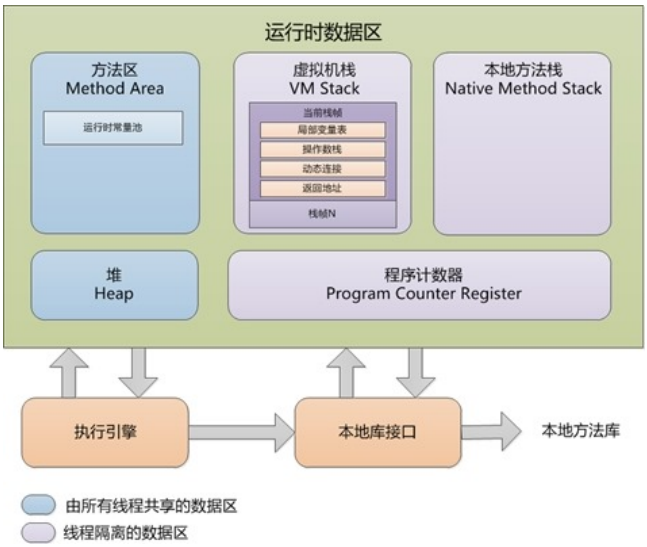

图 9: jvm

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.

3.0.6 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

3.0.7 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.

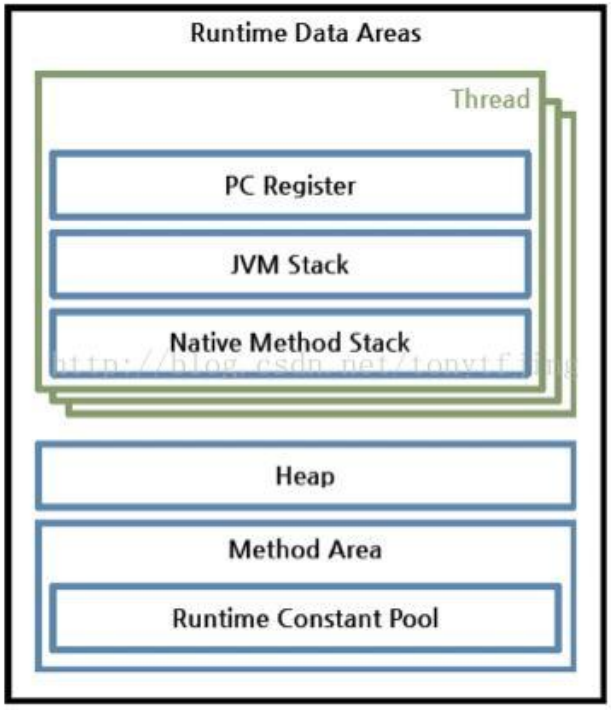

图 10: jvmRunTime

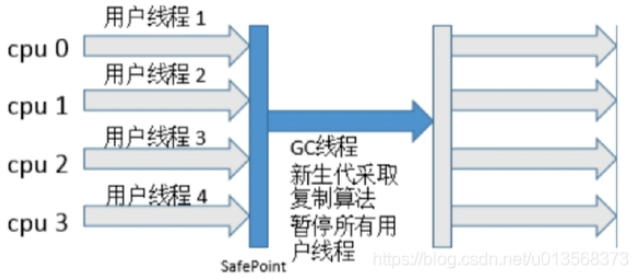

图 11: Serial

3.0.8 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.

3.0.9 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)

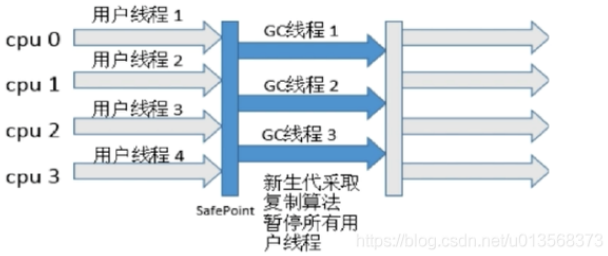

图 12: ParNew

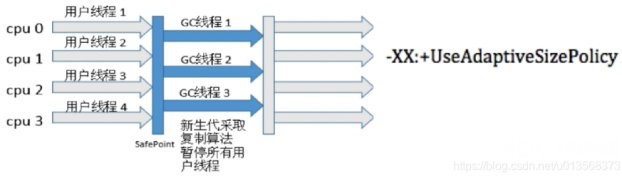

图 13: parallelscavenge

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

(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.

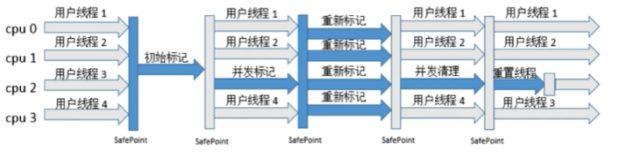

图 14: cms

3.0.10 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村或者, 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

3.0.11 虚拟机性能监控和故障处理工具

jvisualvm 可视化监控.

3.0.12 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

2. 将这个字节流所代表的的静态存储结构转化为方发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, 作为方发渠这个类的各种数据的访问入口

(2) 验证

为了却表 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, 这些变量所使用的的内存都将在**方法区**中进行分配

(4) 解析

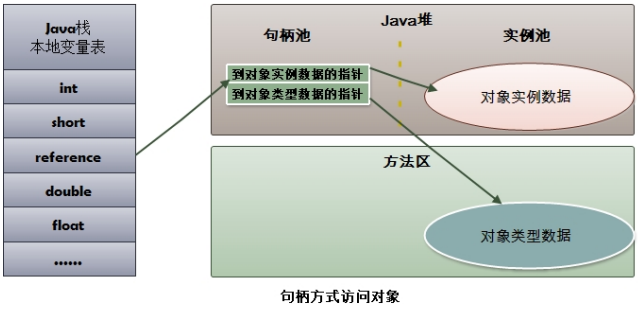

图 15: jvmhandler

虚拟机将常量池内的符号引用替换为直接引用的过程. 主要对类过接口, 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.

3.0.13 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, 代表一个对象再内存中的起始地址
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, 指向固定内存你地址), 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**各自的决堤地址信息, 具体构造如下图所示: **直接指针**

3.0.14 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.

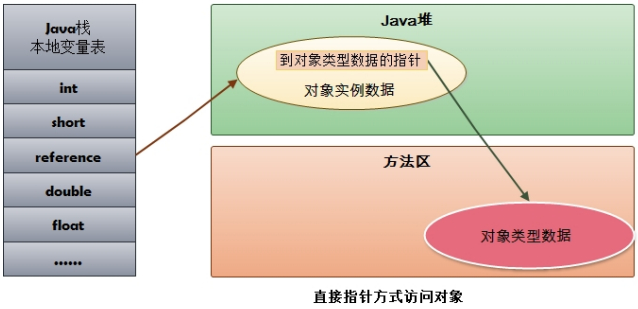

图 16: jvmpoint

引用计数法为每个对象创建一个引用技术, 有对象引用时计数器 +1, 引用被释放是技术-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.

3.0.15 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.

3.0.16 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.

3.0.17 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塔, 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.

4 Redis

4.0.1 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

4.0.2 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

Cluster 模式: 槽分配. 支持多主多从

4.0.3 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额值之间的映射. 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.

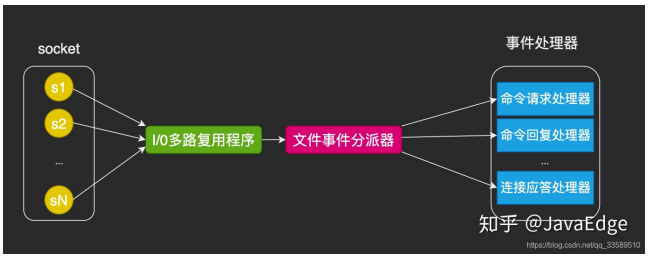

图 17: redisfileevent

4.0.4 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;

4.0.5 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.

表 2: 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)

5. 可以存储二进制, 因为 SDS 不以回车符号进行终止的判定.

4.0.6 redis 字典的底层实现 hashTable 相关问题

1. 解决冲突: 链地址法, 即使用数组 + 链表的方式实现.
2. 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.

4.0.7 压缩链表原理 ziplist

连续内存, 包含多个节点 entry.

4.0.8 zset

本质是舵机链表并有序. 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.

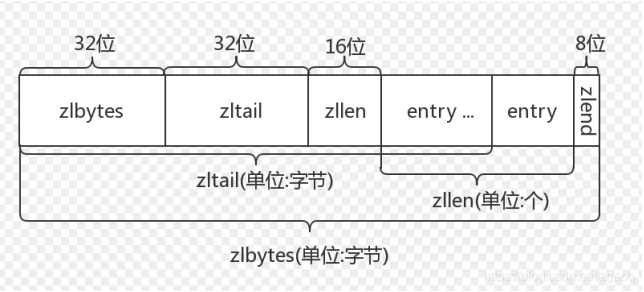

图 18: ziplist

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.

4.0.9 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.

4.0.10 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.

4.0.11 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.

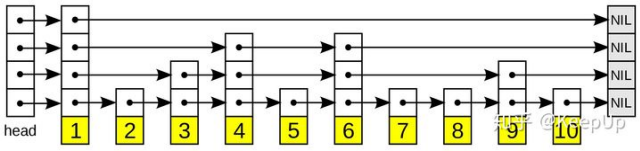

图 19: jumptable

3. 高性能: 使用多路 I/O 复用木星, 非阻塞 IO;

4.0.12 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(lgn), 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.
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.

4.0.13 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.

4.0.14 分布式锁如何使用 redis 实现

1. setnx 命令原子性实现.

4.0.15 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.
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.

4.0.16 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"
```

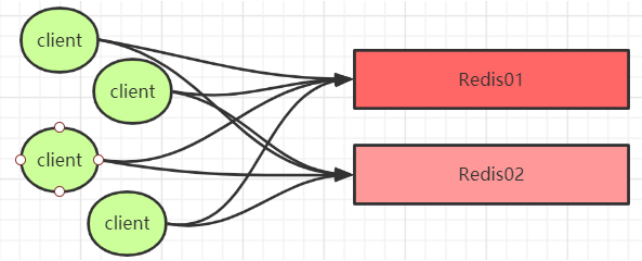

图 20: redis_sharding

```
4        >exec
5        (nil)
```

4.0.17 RedisSharding

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,

4.0.18 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.

4.0.19 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.

4.0.20 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

4.0.21 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;

4.0.22 Redis6.0 为什么要引入多线程呢?

Redis 将所有数据放在内存中, 内存的相应市场大约 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.

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.

4.0.23 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.

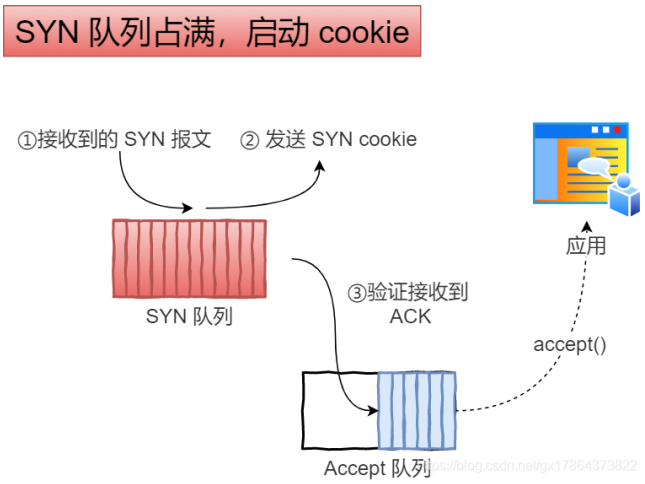

图 21: tcpsynrcvd

4.0.24 Redis 中持久化机制

1.

5 计算机网络

5.0.1 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_syncookies = 1

5.0.2 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

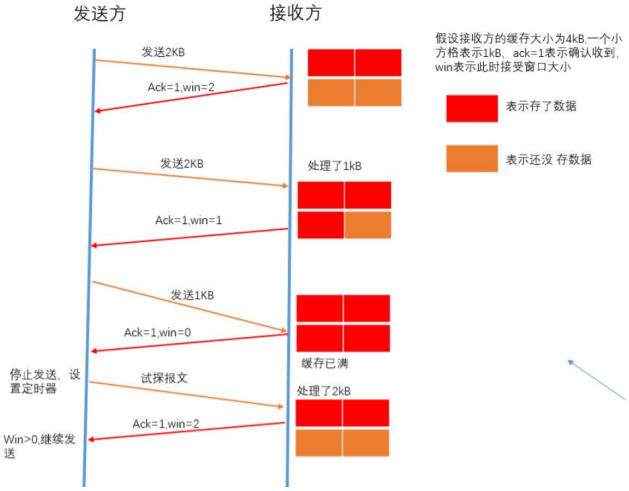

图 22: liuliangkongzhi

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。

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。

5.0.3 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.

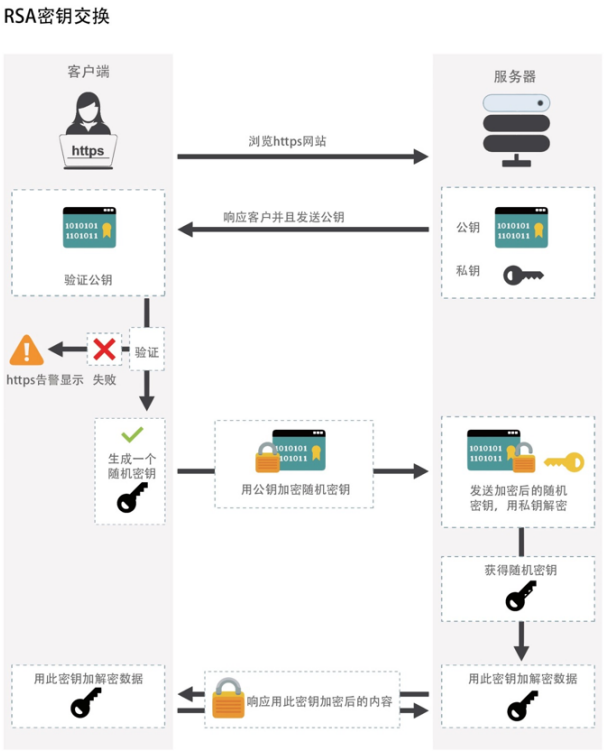

图 23: https

5.0.4 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
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

5.0.5 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

<table>
<tr><th>OSI 体系结构</th><th colspan="2">TCP/IP 协议集</th></tr>
<tr><td>应用层</td><td rowspan="3">应用层</td><td rowspan="3">TELNET、FTP、HTTP、SMTP、DNS 等</td></tr>
<tr><td>表示层</td></tr>
<tr><td>会话层</td></tr>
<tr><td>传输层</td><td>传输层</td><td>TCP、UDP</td></tr>
<tr><td>网络层</td><td>网络层</td><td>IP、ICMP、ARP、RARP</td></tr>
<tr><td>数据链路层</td><td rowspan="2">网络接口层</td><td rowspan="2">各种物理通信网络接口</td></tr>
<tr><td>物理层</td></tr>
</table>

图 24: TCP_IP_OSI

5.0.6 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.

5.0.7 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.

5.0.8 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.

5.0.9 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

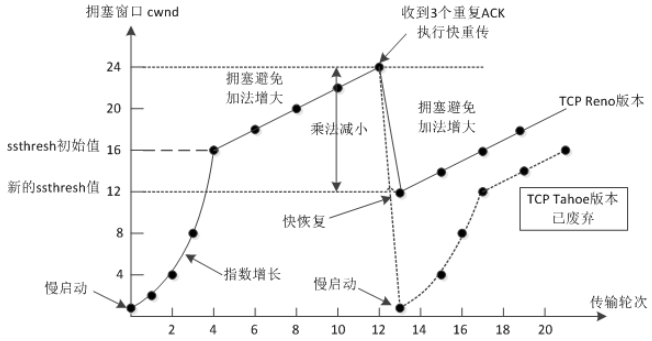

图 25: tcp

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

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, 也就是说由小到大主键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: 发送方只要一脸收到三个重复确认就引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.

5.0.10 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) 用户进行登陆是, 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.

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

5.0.12 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.

5.0.13 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.

6 mysql

6.0.1 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

6.0.2 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

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

undo log: 回滚日志, MVCC 多版本并发. 快照读是 MVCC 的一种体现方式.

6.0.3 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。

6.0.4 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

6.0.5 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.

6.0.6 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

6.0.7 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

6.0.8 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.

6.0.9 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.

6.0.10 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

6.0.11 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.

6.0.12 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. 可以对缓存更好效的应用. 即时其中一个表发生变化, 对其他表的查询缓存依然可以使用.

6.0.13 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 强太多了

另外 inndb 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.

6.0.14 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

6.0.15 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

6.0.16 Innode 加锁

(1) 对于 update,delete 和 insert 语句, Innorb 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.

6.0.17 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.

6.0.18 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.

6.0.19 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

7 操作系统

7.0.1 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

7.0.2 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.

7.0.3 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

7.0.4 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

8 设计模式

8.0.1 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 2018年1月10日
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 2018年1月10日
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 2018年1月10日
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 2018年1月10日
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 2018年1月10日
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 2018年1月10日
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 2018年1月10日
38          * @version v1.0
39          */
40          public static Teacher getTeacher(){
41                  return teacher;
42          }
43
44  }
```

8.0.2 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   /*
2   生产和消费
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
                                        产线程"+Thread.
                                        currentThread().
                                        getName()+"准备休眠
                                        ");
18                                  this.wait();
19                                  System.out.println("生
                                        产线程"+Thread.
                                        currentThread().
                                        getName()+"休眠结束
                                        了");
20                          }
21                          catch (Exception e)
22                          {
23                                  e.printStackTrace();
24                          }
25                  }
26                  this.notify();
27                  data[cnt] = ch;
28                  ++cnt;
29                  System.out.printf("生产线程"+Thread.
                        currentThread().getName()+"正在生产
                        第%d个产品, 该产品是: %c\n", cnt,
                        ch);
30          }
31
32          public synchronized char pop()
33          {
34                  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%d个产品, 该产品是: %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("ss.pop();
96                  }
97          }
98  }
99
100
101 public class TestPC2
102 {
103         public static void main(String[] args)
104         {
105                 SynStack ss = new SynStack();
106                 Producer p = new Producer(ss);
107                 Consumer c = new Consumer(ss);
108
109
110                 Thread t1 = new Thread(p);
111                 t1.setName("1号");
112                 t1.start();
113                 /*Thread t2 = new Thread(p);
114                 t2.setName("2号");
115                 t2.start();*/
116
117                 Thread t6 = new Thread(c);
118                 t6.setName("6号");
119                 t6.start();
120                 Thread t7 = new Thread(c);
121                 t7.setName("7号");
122                 t7.start();
```

```
123             }
124 }
```

8.0.3 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.

9 C++

9.0.1 定义的静态全局变量作用于是

本文件

9.0.2 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

9.0.3 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.

9.0.4 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.

9.0.5 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

9.0.6 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

9.0.7 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

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

malloc/free 是 C/C++ 标准可以函数, 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

9.0.9 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。

9.0.10 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。

9.0.11 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。需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。

9.0.12 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。

9.0.13 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。

9.0.14 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。

9.0.15 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。

9.0.16 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;

9.0.17 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；

(2) 堆：一般由程序员分配和释放，存放动态分配的变量；

(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；

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

9.0.18 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

9.0.19 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

9.0.20 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.

9.0.21 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

9.0.22 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。

9.0.23 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。

9.0.24 RVO 和 NRVO

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

```
Point3d factory()
{
        Point3d x;
        return x;
}
Point3d p = factory();
```

优化成

```
Point3d p;
factory(p);
factory(const Point3d &_result)
{
        Point3d x;
        result.Point3d::Point3d(x);   //复制构造函数还没有吧 x 这个名字优化掉
        return;
}
```

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。

```
Point3d p;
factory(p);
factory(const Point3d &_result)
{
        result.Point3d::Point3d(x); // 直接操作参数, 没有了 x
```

```
6                 return;
7             }
```

9.0.25 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

9.0.26 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.

9.0.27 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, 且不能发现运行是的错误. 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.

9.0.28 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.

9.0.29 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;

9.0.30 auto_ptr 被弃用了

因为可能导致对同一块堆空间多次 delete

9.0.31 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

9.0.32 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

9.0.33 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

9.0.34 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

9.0.35 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.

9.0.36 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。

10 项目解析

10.0.1 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

10.0.2 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.

10.0.3 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，我们受用了向量拟合算法，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，我觉得我写的出来，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. 也就是简单来说, 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.

10.0.4 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 echats 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 echats 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.

11 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.

12 附录: 相关样例

第一个公式

$$F = ma = aa \tag{1}$$

我们可以知道牛顿得出了与物体质量和加速度之间的关系 $F = ma$ 换行公式:

$$\begin{aligned} y &= ax \\ &= bx \end{aligned} \tag{2}$$

12.0.1 小练习

$$v = \frac{x}{y} \tag{3}$$

$$y = e^x \tag{4}$$

$$y = ax^2 + bx + c \tag{5}$$

$$F = G\frac{Mm}{r^2} \tag{6}$$

$$y = 4\pi\frac{\sin x}{\ln x^2} \tag{7}$$

$$y = \sum_{i=1}^{n} x^2 + 1 \tag{8}$$

$$i = \int_1^2 x^2 + \tan x \mathrm{d}x \tag{9}$$

$$A = \begin{bmatrix} 1 & 1 & 3 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \end{bmatrix} \tag{10}$$

12.0.2 三级标题

接下来将开始书写正文

1. 第一个问题：
2. 第二个问题：
3. 第三个问题：

12.0.3 第二个三级标题

12.0.4 第三个三级标题

13 公式的写作

第一个公式

$$F = ma \tag{11}$$

我们可以知道牛顿得出了与物体质量和加速度之间的关系 $F = ma$

换行公式：

$$\begin{aligned} y &= ax \\ &= bx \end{aligned} \tag{12}$$

13.0.1 练习

$$v = \frac{x}{y} \tag{13}$$

$$y = e^x \tag{14}$$

$$y = ax^2 + bx + c \tag{15}$$

$$F = G\frac{Mm}{r^2} \tag{16}$$

$$y = 4\pi\frac{\sin x}{\ln x^2} \tag{17}$$

$$y = \sum_{i=1}^{n} x^2 + 1 \tag{18}$$

现在引用 (18)

$$i = \int_1^2 x^2 + \tan x \mathrm{d}x \tag{19}$$

$$A = \begin{bmatrix} 1 & 1 & 3 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \end{bmatrix} \tag{20}$$

$$A = \begin{cases} x^2 \\ x^2 + x \end{cases} \tag{21}$$

13.0.2 表格的插入

表 3: 标题

指标 1	指标 2	指标 3
居左	居中	居右

表 4: 标题

数据	1	2
甲方	600	700
乙方	800	900