# JavaSE

## 多线程

### 进程与线程的区别？

答:进程是所有线程的集合，每一个线程是进程中的一条执行路径，线程只是一条执行路径。

### 为什么要用多线程？

答:提高程序效率

### 多线程创建方式？

答:继承Thread或Runnable 接口。

### 是继承Thread类好还是实现Runnable接口好？

答:Runnable接口好，因为实现了接口还可以继续继承。继承Thread类不能再继承。

### 你在哪里用到了多线程？

答:主要能体现到多线程提高程序效率。

举例:分批发送短信、迅雷多线程下载等。

### 什么是多线程安全？

答:当多个线程同时共享，同一个**全局变量或静态变量**，做写的操作时，可能会发生数据冲突问题，也就是线程安全问题。做读操作是不会发生数据冲突问题。

### 如何解决多线程之间线程安全问题?

答:使用多线程之间**同步**或使**用锁(lock)**。

### 为什么使用线程同步或使用锁能解决线程安全问题呢？

答:将可能会发生数据冲突问题(线程不安全问题)，**只能让当前一个线程进行执行**。被包裹的代码执行完成后释放锁，让后才能让其他线程进行执行。这样的话就可以解决线程不安全问题。

### 什么是多线程之间同步？

答:当多个线程共享同一个资源,不会受到其他线程的干扰。

### 什么是同步代码块？

答:就是将可能会发生线程安全问题的代码，给包括起来。只能让当前一个线程进行执行，被包裹的代码执行完成之后才能释放所，让后才能让其他线程进行执行。

### 多线程同步的分类？

#### 1.使用同步代码块？

**synchronized(同一个数据){**

**可能会发生线程冲突问题**

**}**

|  |
| --- |
| **private Object mutex = new Object();// 自定义多线程同步锁**  **public void sale() {**  **synchronized (mutex) {**  **if (trainCount > 0) {**  **try {**  **Thread.*sleep*(10);**  **} catch (Exception e) {**  **}**  **System.*out*.println(Thread.*currentThread*().getName() + ",出售 第" + (100 - trainCount + 1) + "张票.");**  **trainCount--; }**  **}**  **}** |

#### 2.使用同步函数

在方法上修饰synchronized 称为同步函数

|  |
| --- |
| **public synchronized void sale() {**  **if (trainCount > 0) {**  **try {**  **Thread.*sleep*(40);**  **} catch (Exception e) {**  **}**  **System.*out*.println(Thread.*currentThread*().getName() + ",出售 第" + (100 - trainCount + 1) + "张票.");**  **trainCount--;**  **}**  **}** |

#### 3.静态同步函数

方法上加上static关键字，使用synchronized 关键字修饰 为静态同步函数

静态的同步函数使用的锁是 该函数所属字节码文件对象

### 同步代码块与同步函数区别？

答:

同步代码使用自定锁(明锁)

同步函数使用this锁

### 同步函数与静态同步函数区别?

注意:有些面试会这样问：例如现在一个静态方法和一个非静态静态怎么实现同步？

答:

同步函数使用this锁

静态同步函数使用字节码文件，也就是类.class

### 什么是多线程死锁?

**答：**

同步中嵌套同步,无法释放锁的资源。

解决办法:同步中尽量不要嵌套同步

### Wait()与Notify ()区别?

**Wait让当前线程有运行状态变为等待状态，和同步一起使用**

**Notify 唤醒现在正在等待的状态,和同步一起使用**

### Wait()与sleep()区别?

**对于sleep()方法，我们首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。**

**sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。**

**在调用sleep()方法的过程中，线程不会释放对象锁。**

**而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备**

**获取对象锁进入运行状态。**

### Lock与Synchronized区别？

**Lock 接口可以尝试非阻塞地获取锁 当前线程尝试获取锁。如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁。  
\*Lock 接口能被中断地获取锁 与 synchronized 不同，获取到锁的线程能够响应中断，当获取到的锁的线程被中断时，中断异常将会被抛出，同时锁会被释放。**

**Lock 接口在指定的截止时间之前获取锁，如果截止时间到了依旧无法获取锁，则返回。**

### Condition用法

 Condition的功能类似于在传统的线程技术中的,Object.wait()和Object.notify()的功能,

**代码:**

|  |
| --- |
| Condition condition = lock.newCondition();res. condition.await(); 类似waitres. Condition. Signal() 类似notify **Signalall notifyALL** |

### 如何停止线程？

**1.  使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。**

**2.  使用stop方法强行终止线程（这个方法不推荐使用，因为stop和suspend、resume一样，也可能发生不可预料的结果）。**

**3.  使用interrupt方法中断线程。 线程在阻塞状态**

### 什么是守护线程

Java中有两种线程，一种是用户线程，另一种是守护线程。

当进程不存在或主线程停止，守护线程也会被停止。

使用setDaemon(true)方法设置为守护线程

### join()方法作用

join作用是让其他线程变为等待，只有当前线程执行完毕后，等待的线程才会被释放。

### 线程三大特性

多线程有三大特性，原子性、可见性、有序性

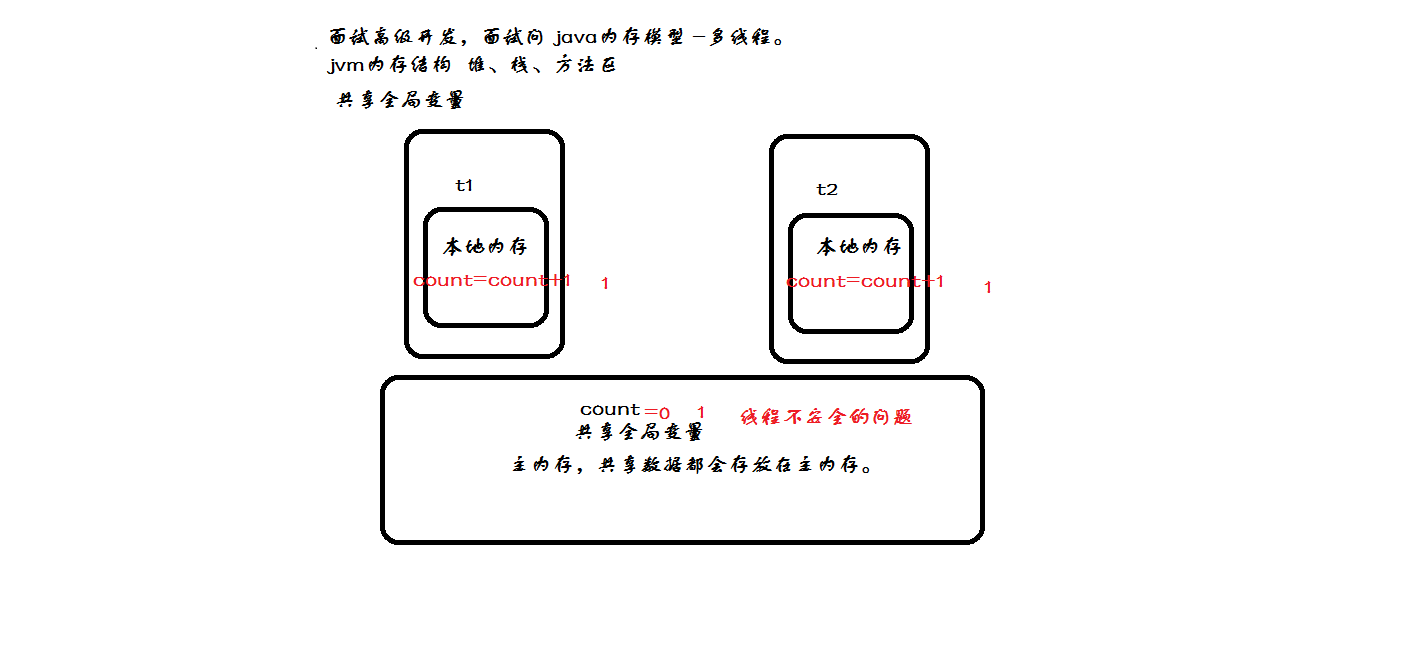
原子性:保证数据一致性，线程安全。

可见性:对另一个线程是否课件

有序性:线程之间执行有顺序

### 说说Java内存模型

共享内存模型指的就是Java内存模型(简称JMM)，JMM决定一个线程对共享变量的写入时,能对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。



### 什么是Volatile作用

Volatile 关键字的作用是变量在多个线程之间可见。

### 什么是AtomicInteger

AtomicInteger原子类

### 什么是ThreadLoca

ThreadLocal提高一个线程的局部变量，访问某个线程拥有自己局部变量。

　当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

ThreadLocal的接口方法

ThreadLocal类接口很简单，只有4个方法，我们先来了解一下：

void set(Object value)设置当前线程的线程局部变量的值。

public Object get()该方法返回当前线程所对应的线程局部变量。

public void remove()将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。

protected Object initialValue()返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

### 什么是线程池？

 线程池是指在初始化一个多线程应用程序过程中创建一个线程集合，然后在需要执行新的任务时重用这些线程而不是新建一个线程。线程池中线程的数量通常完全取决于可用内存数量和应用程序的需求。然而，增加可用线程数量是可能的。线程池中的每个线程都有被分配一个任务，一旦任务已经完成了，线程回到池子中并等待下一次分配任务。

### 线程池作用

基于以下几个原因在多线程应用程序中使用线程是必须的：

  1. 线程池改进了一个应用程序的响应时间。由于线程池中的线程已经准备好且等待被分配任务，应用程序可以直接拿来使用而不用新建一个线程。

  2. 线程池节省了CLR 为每个短生存周期任务创建一个完整的线程的开销并可以在任务完成后回收资源。

  3. 线程池根据当前在系统中运行的进程来优化线程时间片。

  4. 线程池允许我们开启多个任务而不用为每个线程设置属性。

  5. 线程池允许我们为正在执行的任务的程序参数传递一个包含状态信息的对象引用。

  6. 线程池可以用来解决处理一个特定请求最大线程数量限制问题。

### 线程池四种创建方式

Java通过Executors（jdk1.5并发包）提供四种线程池，分别为：  
newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。  
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

### 说说JDK1.5并发包

|  |  |
| --- | --- |
| 名称 | 作用 |
| **Lock** | **锁** |
| **Executors** | **线程池** |
| **ReentrantLock** | **一个可重入的互斥锁定 Lock，功能类似synchronized，但要强大的多。** |
| **Condition** | **Condition的功能类似于在传统的线程技术中的,Object.wait()和Object.notify()的功能,** |
| **ConcurrentHashMap** | **分段HasMap** |
| **AtomicInteger** | **原子类** |
| **BlockingQueue** | **BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景** |
| **ExecutorService** | **执行器服务** |

### 锁的种类

#### 自旋锁

自旋锁是采用让当前线程不停地的在循环体内执行实现的，当循环的条件被其他线程改变时 才能进入临界区。如下

|  |
| --- |
| public class SpinLock {  private AtomicReference<Thread> sign = new AtomicReference<>();  public void lock() {  Thread current = Thread.*currentThread*();  while (!sign.compareAndSet(null, current)) {  }  }  public void unlock() {  Thread current = Thread.*currentThread*();  sign.compareAndSet(current, null);  }  } |

#### 互斥锁

所谓互斥锁, 指的是一次最多只能有一个线程持有的锁. 在jdk1.5之前, 我们通常使用synchronized机制控制多个线程对共享资源**Lock接口及其实现类ReentrantLock**

#### 可重入锁

可重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。  
在JAVA环境下 ReentrantLock 和synchronized 都是 可重入锁

#### 悲观锁

.悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，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（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系 统不会修改数据）。

#### 乐观锁

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库 性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。 而乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如 果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

#### 信号量

信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施, 它负责协调各个线程, 以保证它们能够正确、合理的使用公共资源。

## 集合

## 网络编程

### 什么是Socket？

**Socket就是为网络服务提供的一种机制。**

**通讯的两端都有Sokcet**

**网络通讯其实就是Sokcet间的通讯**

**数据在两个Sokcet间通过IO传输。**

### TCP与UDP在概念上的区别

**udp: a、是面向无连接, 将数据及源的封装成数据包中,不需要建立建立连接**

**b、每个数据报的大小在限制64k内**

**c、因无连接,是不可靠协议**

**d、不需要建立连接,速度快**

**tcp： a、建议连接，形成传输数据的通道.**

**b、在连接中进行大数据量传输，以字节流方式**

**c 通过三次握手完成连接,是可靠协议**

**d 必须建立连接m效率会稍低**

## 设计模式

### 什么是设计模式？

**设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。本章系Java之美[从菜鸟到高手演变]系列之设计模式，我们会以理论与实践相结合的方式来进行本章的学习，希望广大程序爱好者，学好设计模式，做一个优秀的软件工程师！**

### 设计模式的分类？

总体来说设计模式分为三大类：

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

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

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

其实还有两类：并发型模式和线程池模式。用一个图片来整体描述一下：



### 设计模式的六大原则

**1、开闭原则（Open Close Principle）**

开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**2、里氏代换原则（Liskov Substitution Principle）**

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

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

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

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

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

**5、迪米特法则（最少知道原则）（Demeter Principle）**

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

**6、合成复用原则（Composite Reuse Principle）**

原则是尽量使用合成/聚合的方式，而不是使用继承。

### 单例模式

#### 什么是单例模式？

单例保证一个对象JVM中只能有一个实例,常见单例 懒汉式、饿汉式

什么是懒汉式,就是需要的才会去实例化,线程不安全。

什么是饿汉式,就是当class文件被加载的时候，初始化，天生线程安全。

#### 单例写法

**懒汉式代码**

|  |
| --- |
| **class SingletonTest {**  **public static void main(String[] args) {**  **Singleton sl1 = Singleton.*getSingleton*();**  **Singleton sl2 = Singleton.*getSingleton*();**  **System.*out*.println(sl1 == sl2);**  **}**  **}**  **public class Singleton {**  **// 当需要的才会被实例化**  **private static Singleton *singleton*;**  **private Singleton() {**  **}**  **synchronized public static Singleton getSingleton() {**  **if (*singleton* == null) {**  ***singleton* = new Singleton();**  **}**  **return *singleton*;**  **}**  **}** |

**饿汉式代码**

|  |
| --- |
| **class SingletonTest1 {**  **public static void main(String[] args) {**  **Singleton1 sl1 = Singleton1.*getSingleton*();**  **Singleton1 sl2 = Singleton1.*getSingleton*();**  **System.*out*.println((sl1 == sl2)+"-");**  **}**  **}**  **public class Singleton1 {**  **//当class 文件被加载初始化**  **private static Singleton1 *singleton* = new Singleton1();**  **private Singleton1() {**  **}**  **public static Singleton1 getSingleton() {**  **return *singleton*;**  **}**  **}** |

### 工厂模式

#### 什么是工厂模式？

**实现创建者和调用者分离**

#### 简单工厂代码

|  |
| --- |
| **public interface Car {**  **public void run();**  **}**  **public class AoDi implements Car {**  **@Override**  **public void run() {**  **System.*out*.println("奥迪....");**  **}**  **}**  **public interface Car {**  **public void run();**  **}** |

|  |
| --- |
| **public** **class** CarFactory {  **static** **public** Car createCar(String carName) {  Car car = **null**;  **if** (carName.equals("奥迪")) {  car = **new** AoDi();  } **else** **if** (carName.equals("奔驰")) {  car = **new** BenChi();  }  **return** car;  }  **public** **static** **void** main(String[] args) {  Car car1 = CarFactory.*createCar*("奥迪");  Car car2 = CarFactory.*createCar*("奔驰");  car1.run();  car2.run();  }  } |

#### 工厂方法

|  |
| --- |
| public interface Car {  public void run();  }  public class AoDi implements Car {  @Override  public void run() {  System.*out*.println("奥迪....");  }  }  public class BenChi implements Car {  @Override  public void run() {  System.*out*.println("奔驰....");  }  } |

|  |
| --- |
| **public class AoDiChiFactory {**  **static public Car createCar() {**  **return new AoDi();**  **}**  **}**  **public interface BenChiFactory {**  **static public Car createCar() {**  **return new BenChi();**  **}**  **}**  **public class Main {**  **public static void main(String[] args) {**  **Car c1 = AoDiChiFactory.*createCar*();**  **Car c2 = BenChiFactory.*createCar*();**  **c1.run();**  **c2.run();**  **}**  **}** |

### 代理模式

#### 什么是代理？

通过代理控制对象的访问,可以详细访问某个对象的方法，在这个方法调用处理，或调用后处理。既(AOP微实现) ,AOP核心技术面向切面编程。



#### 代理应用场景

安全代理 可以屏蔽真实角色

远程代理 远程调用代理类RMI

延迟加载 先加载轻量级代理类,真正需要在加载真实

#### 代理的分类

静态代理(静态定义代理类)

动态代理(动态生成代理类)

Jdk自带动态代理

Cglib 、javaassist（字节码操作库）

#### 静态代理

静态代理需要自己生成代理类

|  |
| --- |
| **public** **class** XiaoMing **implements** Hose {  @Override  **public** **void** mai() {  System.***out***.println("我是小明,我要买房啦!!!!haha ");  }  }  **class** Proxy **implements** Hose {  **private** XiaoMing xiaoMing;  **public** Proxy(XiaoMing xiaoMing) {  **this**.xiaoMing = xiaoMing;  }  **public** **void** mai() {  System.***out***.println("我是中介 看你买房开始啦!");  xiaoMing.mai();  System.***out***.println("我是中介 看你买房结束啦!");  }  **public** **static** **void** main(String[] args) {  Hose proxy = **new** Proxy(**new** XiaoMing());  proxy.mai();  }  } |

#### JDK动态代理(不需要生成代理类)

实现InvocationHandler 就可以了。

|  |
| --- |
| **public** **interface** Hose {  /\*\*  \*  \* **@methodDesc**: 功能描述:(买房代理)  \* **@author**: 余胜军  \* **@param**:  \* **@createTime**:2017年8月27日 上午2:54:34  \* **@returnType**: void  \* **@copyright**:上海每特教育科技有限公司  \*/  **public** **void** mai();  }  **public** **class** XiaoMing **implements** Hose {  @Override  **public** **void** mai() {  System.***out***.println("我是小明,我要买房啦!!!!haha ");  }  } |

|  |
| --- |
| **public** **class** JDKProxy **implements** InvocationHandler {  **private** Object tarjet;  **public** JDKProxy(Object tarjet) {  **this**.tarjet = tarjet;  }  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  System.***out***.println("我是房产中介.....开始监听你买房啦!");  Object oj = method.invoke(tarjet, args);  System.***out***.println("我是房产中介.....结束监听你买房啦!");  **return** oj;  }  }  **class** Test222 {  **public** **static** **void** main(String[] args) {  XiaoMing xiaoMing = **new** XiaoMing();  JDKProxy jdkProxy = **new** JDKProxy(xiaoMing);  Hose hose=(Hose) Proxy.*newProxyInstance*(xiaoMing.getClass().getClassLoader(), xiaoMing.getClass().getInterfaces(), jdkProxy);  hose.mai();  }  } |

#### CGLIB动态代理

实现

|  |
| --- |
| **import java.lang.reflect.Method;**  **import net.sf.cglib.proxy.Enhancer;**  **import net.sf.cglib.proxy.MethodInterceptor;**  **import net.sf.cglib.proxy.MethodProxy;**  **public class Cglib implements MethodInterceptor {**  **@Override**  **public Object intercept(Object o, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {**  **System.*out*.println("我是买房中介 ， 开始监听你买房了....");**  **Object invokeSuper = methodProxy.invokeSuper(o, args);**  **System.*out*.println("我是买房中介 ， 开结束你买房了....");**  **return invokeSuper;**  **}**  **}**  **class Test22222 {**  **public static void main(String[] args) {**  **Cglib cglib = new Cglib();**  **Enhancer enhancer = new Enhancer();**  **enhancer.setSuperclass(XiaoMing.class);**  **enhancer.setCallback(cglib);**  **Hose hose = (Hose) enhancer.create();**  **hose.mai();**  **}**  **}** |

#### CGLIB与JDK动态代理区别

jdk动态代理是由[Java](http://lib.csdn.net/base/java)内部的反射机制来实现的，cglib动态代理底层则是借助asm来实现的。总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题）。还有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。

注:asm其实就是java字节码控制.

## 其他

### 什么是注解？

**Jdk1.5新增新技术，注解。很多框架为了简化代码，都会提供有些注解。可以理解为插件，是代码级别的插件，在类的方法上写：@XXX，就是在代码上插入了一个插件。**

**注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。**

**注解分类：内置注解(也成为元注解 jdk 自带注解)、自定义注解（Spring框架）**

### 如何定义一个注解？

**代码:**

**使用@interface 定义注解。**

|  |
| --- |
| @Target(value = { ElementType.***METHOD***, ElementType.***TYPE*** })  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** OneAnnotation {  **int** beanId() **default** 0;  String className() **default** "";  String[]arrays();  } |

### 什么是数据交换格式？

**客户端与服务器常用数据交换格式xml、json、html**

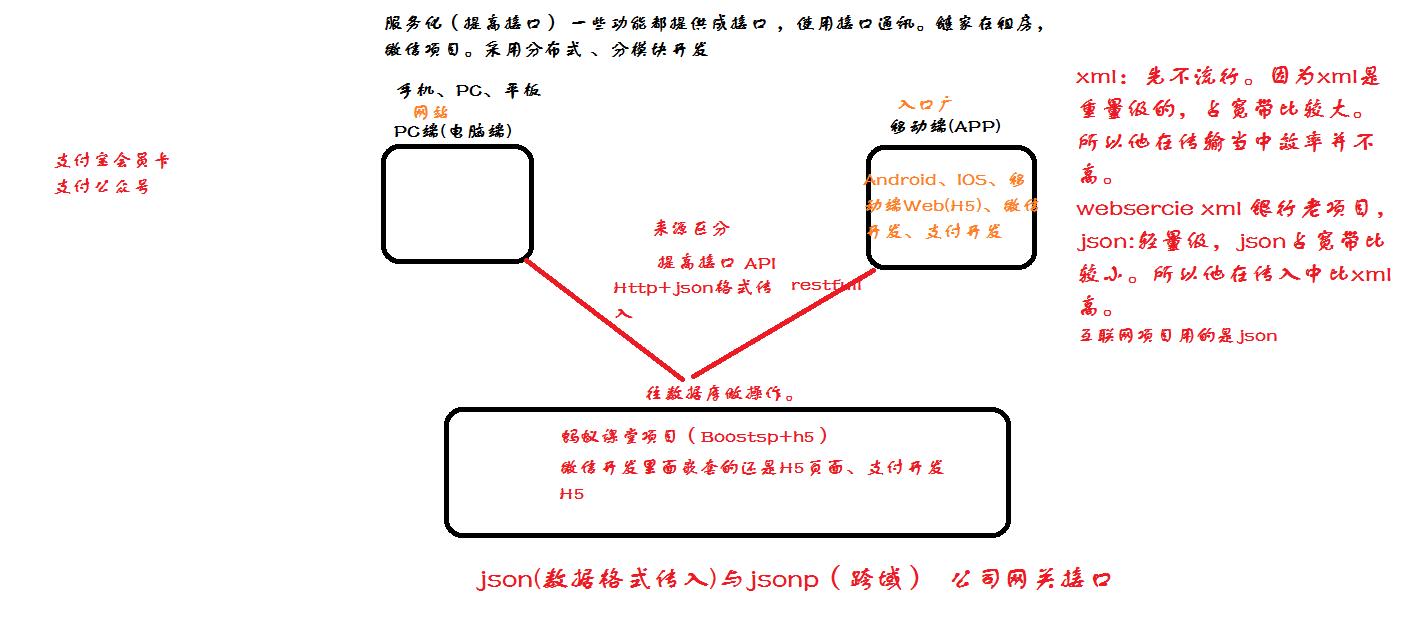
### 数据交换格式用场景

**移动端(安卓、IOS)通讯方式采用http协议+JSON格式 走restful风格。**

**很多互联网项目都采用Http协议+JSON**

**因为xml比较重WebService服务采用http+xml格式 银行项目使用比较多**

**同学们可以思考下？移动端和PC端服务器是接口是怎么设计的？**



### JSON解析框架有哪些？

**fastjson(阿里)、gson(谷歌)、jackson(SpringMVC自带)**

### XML解析方式？

**Dom4j、Sax、Pull**

### Dom4j与Sax区别

**dom4j不适合大文件的解析，因为它是一下子将文件加载到内存中，所以有可能出现内存溢出，sax是基于事件来对xml进行解析的，所以他可以解析大文件的xml，也正是因为如此，所以dom4j可以对xml进行灵活的增删改查和导航，而sax没有这么强的灵活性，所以sax经常是用来解析大型xml文件，而要对xml文件进行一些灵活（crud）操作就用dom4j。**

### XML与JSON区别

**Xml是重量级数据交换格式，占宽带比较大。**

**JSON是轻量级交换格式，xml占宽带小。**

**所有很多互联网公司都会使用json作为数据交换格式**

**很多银行项目，大多数还是在使用xml。**

### 什么是Java反射

就是正在运行，动态获取这个类的所有信息。

### 反射机制的作用

  1，反编译：.class-->.java

   2．通过反射机制访问java对象的属性，方法，构造方法等；

### 反射机制的应用场景

Jdbc 加载驱动-----

Spring ioc

框架

### 反射机制获取类有三种方法

|  |
| --- |
| **//第一种方式：**  **Classc1 = Class.*forName*("Employee");**  **//第二种方式：**  **//java中每个类型都有class 属性.**  **Classc2 = Employee.class;**    **//第三种方式：**  **//java语言中任何一个java对象都有getClass 方法**  **Employeee = new Employee();**  **Classc3 = e.getClass(); //c3是运行时类 (e的运行时类是Employee)** |

### 反射创建对象的方式

|  |
| --- |
| **Class<?> forName = Class.*forName*("com.itmayiedu.entity.User");**  **// 创建此Class 对象所表示的类的一个新实例 调用了User的无参数构造方法.**  **Object newInstance = forName.newInstance();** |

实例化有参构造函数

|  |
| --- |
| **Class<?> forName = Class.*forName*("com.itmayiedu.entity.User");**  **Constructor<?> constructor = forName.getConstructor(String.class, String.class);**  **User newInstance = (User) constructor.newInstance("123", "123");** |

### 反射创建api

|  |  |
| --- | --- |
| 方法名称 | 作用 |
| getDeclaredMethods [] | 获取该类的所有方法 |
| getReturnType() | 获取该类的返回值 |
| getParameterTypes() | 获取传入参数 |
| getDeclaredFields() | 获取该类的所有字段 |
| setAccessible | 允许访问私有成员 |

### 使用反射为类私有属性赋值

|  |
| --- |
| **// 获取当前类class地址**  **Class<?> forName = Class.*forName*("com.itmayiedu.entity.User");**  **// 使用反射实例化对象 无参数构造函数**  **Object newInstance = forName.newInstance();**  **// 获取当前类的 userId字段**  **Field declaredField = forName.getDeclaredField("userId");**  **// 允许操作私有成员**  **declaredField.setAccessible(true);**  **// 设置值**  **declaredField.set(newInstance, "123");**  **User user = (User) newInstance;**  **System.*out*.println(user.getUserId());** |

## JVM参数调优

### Java虚拟机原理

**所谓虚拟机，就是一台虚拟的机器。他是一款软件，用来执行一系列虚拟计算指令，大体上虚拟机可以分为**

**系统虚拟机和程序虚拟机， 大名鼎鼎的Visual Box、Vmare就属于系统虚拟机，他们完全是对物理计算的仿真，**

**提供了一个可以运行完整操作系统的软件平台。**

**程序虚拟机典型代码就是Java虚拟机，它专门为执行单个计算程序而计算，在Java虚拟机中执行的指令我们成为Java**

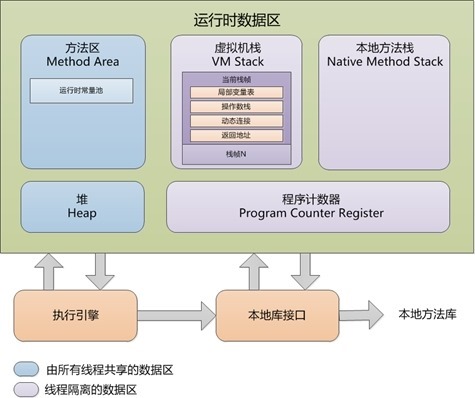
**自己码指令。无论是系统虚拟机还是程序虚拟机，在上面运行的软件都被限制于虚拟机提供的资源中。**

**Java发展至今，出现过很多虚拟机，做初Sun使用的一款叫ClassIc的Java虚拟机，到现在引用最广泛的是HotSpot虚拟**

**机，除了Sum意外，还有BEA的Jrockit，目前Jrockit和HostSopt都被oralce收入旗下，大有整合的趋势。**

### Java内存结构





1. **类加载子系统:负责从文件系统或者网络加载Class信息，加载的信息存放在一块称之方法区的内存空间。**
2. **方法区:就是存放类的信息、常量信息、常量池信息、包括字符串字面量和数字常量等。**
3. **Java堆：在Java虚拟机启动的时候建立Java堆，它是Java程序最主要的内存工作区域，几乎所有的对象实例都存放到**

**Java堆中，堆空间是所有线程共享。**

1. **直接内存：JavaNio库允许Java程序直接内存，从而提高性能，通常直接内存速度会优于Java堆。读写频繁的场合可能会考虑使用。**
2. **每个虚拟机线程都有一个私有栈，一个线程的Java栈在线程创建的时候被创建，Java栈保存着局部变量、方法参数、同事Java的方法调用、**

**返回值等。**

1. **本地方法栈，最大不同为本地方法栈用于本地方法调用。Java虚拟机允许Java直接调用本地方法（通过使用C语言写）**
2. **垃圾收集系统是Java的核心，也是不可少的，Java有一套自己进行垃圾清理的机制，开发人员无需手工清理，下一节课详细讲。**
3. **PC（Program Couneter）寄存器也是每个线程私有的空间， Java虚拟机会为每个线程创建PC寄存器，在任意时刻，**

**一个Java线程总是在执行一个方法，这个方法称为当前方法，如果当前方法不是本地方法，PC寄存器总会执行当前正在被执行的指令，**

**如果是本地方法，则PC寄存器值为Underfined，寄存器存放如果当前执行环境指针、程序技术器、操作栈指针、计算的变量指针等信息。**

1. **虚拟机核心的组件就是执行引擎，它负责执行虚拟机的字节码，一般户先进行编译成机器码后执行。**

### 堆、栈、方法区概念区别

#### Java堆

**堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。**

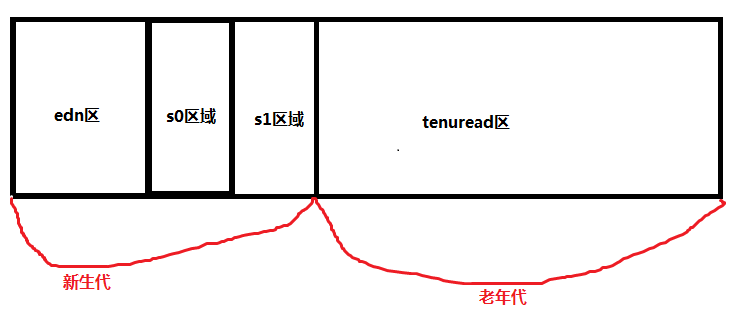
**根据垃圾回收机制的不同，Java堆有可能拥有不同的结构，最为常见的就是将整个Java堆分为**

**新生代和老年代。其中新声带存放新生的对象或者年龄不大的对象，老年代则存放老年对象。**

**新生代分为den区、s0区、s1区，s0和s1也被称为from和to区域，他们是两块大小相等并且可以互相角色的空间。**

**绝大多数情况下，对象首先分配在eden区，在新生代回收后，如果对象还存活，则进入s0或s1区，之后每经过一次**

**新生代回收，如果对象存活则它的年龄就加1，对象达到一定的年龄后，则进入老年代。**



#### Java栈

**Java栈是一块线程私有的空间，一个栈，一般由三部分组成:局部变量表、操作数据栈和帧数据区**

**局部变量表：用于报错函数的参数及局部变量**

**操作数栈：主要保存计算过程的中间结果，同时作为计算过程中的变量临时的存储空间。**

**帧数据区:除了局部变量表和操作数据栈以外，栈还需要一些数据来支持常量池的解析，这里帧数据区保存着**

**访问常量池的指针，方便计程序访问常量池，另外当函数返回或出现异常时卖虚拟机子必须有一个异常处理表，方便发送异常**

**的时候找到异常的代码，因此异常处理表也是帧数据区的一部分。**

****

#### Java方法区

**Java方法区和堆一样，方法区是一块所有线程共享的内存区域，他保存系统的类信息。**

**比如类的字段、方法、常量池等。方法区的大小决定系统可以保存多少个类。如果系统**

**定义太多的类，导致方法区溢出。虚拟机同样会抛出内存溢出的错误。方法区可以理解**

**为永久区。**

### 虚拟机参数配置

#### 什么是虚拟机参数配置

**在虚拟机运行的过程中，如果可以跟踪系统的运行状态，那么对于问题的故障**

**排查会有一定的帮助，为此，在虚拟机提供了一些跟踪系统状态的参数，使用**

**给定的参数执行Java虚拟机，就可以在系统运行时打印相关日志，用于分析实际**

**问题。我们进行虚拟机参数配置，其实就是围绕着堆、栈、方法区、进行配置。**

**你说下 你熟悉那些jvm参数调优**

#### 堆的参数配置

**-XX:+PrintGC    每次触发GC的时候打印相关日志**

**-XX:+UseSerialGC 串行回收**

**-XX:+PrintGCDetails  更详细的GC日志**

**-Xms    堆初始值**

**-Xmx    堆最大可用值**

**-Xmn    新生代堆最大可用值**

**-XX:SurvivorRatio 用来设置新生代中eden空间和from/to空间的比例.**

**含以-XX:SurvivorRatio=eden/from=den/to**

**总结:在实际工作中，我们可以直接将初始的堆大小与最大堆大小相等，**

**这样的好处是可以减少程序运行时垃圾回收次数，从而提高效率。**

**-XX:SurvivorRatio 用来设置新生代中eden空间和from/to空间的比例.**

### 设置最大堆内存

**参数: -Xms5m -Xmx20m -XX:+PrintGCDetails -XX:+UseSerialGC -XX:+PrintCommandLineFlags**

|  |
| --- |
| **/\*\***  **\* jvm参数设置**  **\***  **\* @author Administrator**  **\***  **\*/**  **public class JvmDemo01 {**  **public static void main(String[] args) throws InterruptedException {**  **byte[] b1 = new byte[1 \* 1024 \* 1024];**  **System.*out*.println("分配了1m");**  ***jvmInfo*();**  **Thread.*sleep*(3000);**  **byte[] b2 = new byte[4 \* 1024 \* 1024];**  **System.*out*.println("分配了4m");**  **Thread.*sleep*(3000);**  ***jvmInfo*();**  **}**  **/\*\***  **\* 转换为m**  **\***  **\* @param maxMemory**  **\* @return**  **\*/**  **static private String toM(long maxMemory) {**  **float num = (float) maxMemory / (1024 \* 1024);**  **DecimalFormat df = new DecimalFormat("0.00");// 格式化小数**  **String s = df.format(num);// 返回的是String类型**  **return s;**  **}**  **static private void jvmInfo() {**  **// 最大内存**  **long maxMemory = Runtime.*getRuntime*().maxMemory();**  **System.*out*.println("maxMemory:" + maxMemory + ",转换为M:" + *toM*(maxMemory));**  **// 当前空闲内存**  **long freeMemory = Runtime.*getRuntime*().freeMemory();**  **System.*out*.println("freeMemory:" +freeMemory+",转换为M:"+*toM*(freeMemory));**  **// 已经使用内存**  **long totalMemory = Runtime.*getRuntime*().totalMemory();**  **System.*out*.println("totalMemory:" +totalMemory+",转换为M"+*toM*(totalMemory));**  **}**  **}** |

### 设置新生代与老年代优化参数

**-Xmn    新生代大小，一般设为整个堆的1/3到1/4左右**

**-XX:SurvivorRatio    设置新生代中eden区和from/to空间的比例关系n/1**

#### 设置新生代比例参数

**参数: -Xms20m -Xmx20m -Xmn1m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC**

|  |
| --- |
| **public class JvmDemo02 {**  **public static void main(String[] args) {**  **//-Xms20m -Xmx20m -Xmn1m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC**  **byte [] b = null;**  **for (int i = 0; i < 10; i++) {**  **b =new byte[1\*1024\*1024];**  **}**    **}**    **}** |

#### 设置新生与老年代代参数

-Xms20m -Xmx20m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC

-Xms20m -Xmx20m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC

**-XX:NewRatio=2**

**总结:不同的堆分布情况，对系统执行会产生一定的影响，在实际工作中，**

**应该根据系统的特点做出合理的配置，基本策略：尽可能将对象预留在新生代，**

**减少老年代的GC次数。**

**除了可以设置新生代的绝对大小(-Xmn),可以使用(-XX:NewRatio)设置新生代和老年**

**代的比例:-XX:NewRatio=老年代/新生代**

#### 内存溢出解决办法

##### 设置堆内存大小

**错误原因: java.lang.OutOfMemoryError: Java heap space**

**解决办法:设置堆内存大小 -Xms1m -Xmx70m -XX:+HeapDumpOnOutOfMemoryError**

|  |
| --- |
| **public static void main(String[] args) throws InterruptedException {**  **List<Object> list = new ArrayList<>();**  **Thread.*sleep*(3000);**  ***jvmInfo*();**  **for (int i = 0; i < 10; i++) {**  **System.*out*.println("i:"+i);**  **Byte [] bytes= new Byte[1\*1024\*1024];**  **list.add(bytes);**  ***jvmInfo*();**  **}**  **System.*out*.println("添加成功...");**  **}** |

##### 设置栈内存大小

**错误原因: java.lang.StackOverflowError**

**栈溢出 产生于递归调用，循环遍历是不会的，但是循环方法里面产生递归调用， 也会发生栈溢出。**

**解决办法:设置线程最大调用深度**

**-Xss5m 设置最大调用深度**

|  |
| --- |
| **public class JvmDemo04 {**  **private static int *count*;**  **public static void count(){**  **try {**  ***count*++;**  ***count*();**  **} catch (Throwable e) {**  **System.*out*.println("最大深度:"+*count*);**  **e.printStackTrace();**  **}**  **}**  **public static void main(String[] args) {**  ***count*();**  **}**  **}** |

#### Tomcat内存溢出在catalina.sh 修改JVM堆内存大小 JAVA\_OPTS="-server -Xms800m -Xmx800m -XX:PermSize=256m -XX:MaxPermSize=512m -XX:MaxNewSize=512m"

### 内存溢出与内存泄露的区别

内存溢出 out of memory，是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个integer,但给它存了long才能存下的数，那就是内存溢出。

内存泄露 memory leak，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

memory leak会最终会导致out of memory！

内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。   
  
    内存泄漏是指你向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。一个盘子用尽各种方法只能装4个果子，你装了5个，结果掉倒地上不能吃了。这就是溢出！比方说栈，栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。就是分配的内存不足以放下数据项序列,称为内存溢出.   
  
   以发生的方式来分类，内存泄漏可以分为4类：   
  
1. 常发性内存泄漏。发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。   
2. 偶发性内存泄漏。发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。   
3. 一次性内存泄漏。发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块仅且一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存，所以内存泄漏只会发生一次。   
4. 隐式内存泄漏。程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。   
  
从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为一般的用户，根本感觉不到内存泄漏的存在。真正有危害的是内存泄漏的堆积，这会最终消耗尽系统所有的内存。从这个角度来说，一次性内存泄漏并没有什么危害，因为它不会堆积，而隐式内存泄漏危害性则非常大，因为较之于常发性和偶发性内存泄漏它更难被检测到

### JVM参数调优总结

    在JVM启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任何设置JVM会工作的很好，但对一些配置很好的Server和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到一些目标：

* GC的时间足够的小
* GC的次数足够的少
* 发生Full GC的周期足够的长

  前两个目前是相悖的，要想GC时间小必须要一个更小的堆，要保证GC次数足够少，必须保证一个更大的堆，我们只能取其平衡。

   （1）针对JVM堆的设置，一般可以通过-Xms -Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值  
   （2）年轻代和年老代将根据默认的比例（1：2）分配堆内存，可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代，比如年轻代，通过 -XX:newSize -XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize设置为同样大小

   （3）年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响

* 更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的Full GC
* 更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短；大的年老代会减少Full GC的频率
* 如何选择应该依赖应用程序对象生命周期的分布情况：如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：（A）本着Full GC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理 （B）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响Full GC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间

## 垃圾回收机制

### 垃圾回收机制概述

　Java语言中一个显著的特点就是引入了垃圾回收机制，使c++程序员最头疼的内存管理的问题迎刃而解，它使得Java程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java中的对象不再有“作用域”的概念，只有对象的引用才有“作用域”。垃圾回收可以有效的防止内存泄露，有效的使用空闲的内存。

　　ps:内存泄露是指该内存空间使用完毕之后未回收，在不涉及复杂数据结构的一般情况下，Java 的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度，我们有时也将其称为“对象游离”。

### 垃圾回收简要过程

**这里必须点出一个很重要的误区：不可达的对象并不会马上就会被直接回收，而是至少要经过两次标记的过程。   
        第一次被标记过的对象，会检查该对象是否重写了finalize()方法。如果重写了该方法，则将其放入一个F-Query队列中，否则，直接将对象加入“即将回收”集合。在第二次标记之前，F-Query队列中的所有对象会逐个执行finalize()方法，但是不保证该队列中所有对象的finalize()方法都能被执行，这是因为JVM创建一个低优先级的线程去运行此队列中的方法，很可能在没有遍历完之前，就已经被剥夺了运行的权利。那么运行finalize()方法的意义何在呢？这是对象避免自己被清理的最后手段：如果在执行finalize()方法的过程中，使得此对象重新与GC Roots引用链相连，则会在第二次标记过程中将此对象从F-Query队列中清除，避免在这次回收中被清除，恢复成了一个“正常”的对象。但显然这种好事不能无限的发生，对于曾经执行过一次finalize()的对象来说，之后如果再被标记，则不会再执行finalize()方法，只能等待被清除的命运。   
        之后，GC将对F-Queue中的对象进行第二次小规模的标记，将队列中重新与GC Roots引用链恢复连接的对象清除出“即将回收”集合。所有此集合中的内容将被回收。**

### 手动GC回收

|  |
| --- |
| **public** **class** JVMDemo05 {  **public** **static** **void** main(String[] args) {  JVMDemo05 jvmDemo05 = **new** JVMDemo05();  //jvmDemo05 = null;  System.*gc*();  }  **protected** **void** finalize() **throws** Throwable {  System.***out***.println("gc在回收对象...");  }  } |

### finalize作用

**Java技术使用finalize()方法在垃圾收集器将对象从内存中清除出去前，做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在Object类中定义的，因此所有的类都继承了它。子类覆盖finalize()方法以整理系统资源或者执行其他清理工作。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。**

### 垃圾回收机制算法

#### 引用计数法

##### 概述

**给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器都为0的对象就是不再被使用的，垃圾收集器将回收该对象使用的内存。**

##### 优缺点

**优点：**

**引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。**

**缺点：**

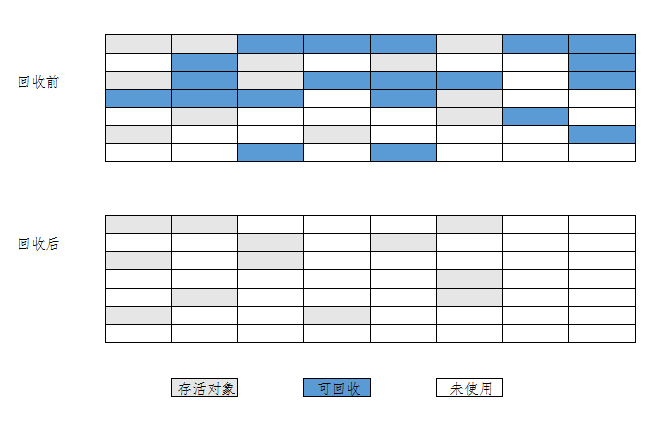
**无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0.而且每次加减非常浪费内存。**

#### 标记清除算法

**标记-清除（Mark-Sweep）算法顾名思义，主要就是两个动作，一个是标记，另一个就是清除。**

**标记就是根据特定的算法（如：引用计数算法，可达性分析算法等）标出内存中哪些对象可以回收，哪些对象还要继续用。**

**标记指示回收，那就直接收掉；标记指示对象还能用，那就原地不动留下。**

****

**缺点**

1. **标记与清除效率低;**
2. **清除之后内存会产生大量碎片；**

**所以碎片这个问题还得处理，怎么处理，看标记-整理算法。**

#### 复制算法

**S0和s1将可用内存按容量分成大小相等的两块，每次只使用其中一块，当这块内存使用完了，就将还存活的对象复制到另一块内存上去，然后把使用过的内存空间一次清理掉。这样使得每次都是对其中一块内存进行回收，内存分配时不用考虑内存碎片等复杂情况，只需要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。**

**复制算法的缺点显而易见，可使用的内存降为原来一半。**

**复制算法用于在新生代垃圾回收**

#### 标记-压缩算法

标记压缩法在标记清除基础之上做了优化，把存活的对象压缩到内存一端,而后进行垃圾清理。(java中老年代使用的就是标记压缩法)

#### 分代收集算法

根据内存中对象的存活周期不同，将内存划分为几块，java的虚拟机中一般把内存划分为新生代和年老代，当新创建对象时一般在新生代中分配内存空间，当新生代垃圾收集器回收几次之后仍然存活的对象会被移动到年老代内存中，当大对象在新生代中无法找到足够的连续内存时也直接在年老代中创建。

对于新生代和老年代来说,新生代回收频率很高,但是每次回收耗时很短,而老年代回收频率较低,但是耗时会相对较长,所以应该尽量减少老年代的GC.

#### 为什么老年代使用标记压缩、新生代使用复制算法。

#### 垃圾回收时的停顿现象

垃圾回收的任务是识别和回收垃圾对象进行内存清理，为了让垃圾回收器可以更高效的执行，大部分情况下，会要求系统进如一个停顿的状态。停顿的目的是为了终止所有的应用线程，只有这样的系统才不会有新垃圾的产生。同时停顿保证了系统状态在某一个瞬间的一致性，也有利于更好的标记垃圾对象。因此在垃圾回收时，都会产生应用程序的停顿。

### 垃圾收集器

#### 什么是Java垃圾回收器

Java垃圾回收器是Java虚拟机(JVM)的三个重要模块(另外两个是解释器和多线程机制)之一，为应用程序提供内存的**自动分配(Memory Allocation)、自动回收(Garbage Collect)**功能，这两个操作都发生在Java堆上(一段内存快)。某一个时点，一个对象如果有一个以上的引用(Rreference)指向它，那么该对象就为活着的(Live)，否则死亡(Dead)，视为垃圾，可被垃圾回收器回收再利用。**垃圾回收操作需要消耗CPU、线程、时间等资源**，所以容易理解的是**垃圾回收操作不是实时的发生**(对象死亡马上释放)，当内存消耗完或者是达到某一个指标(Threshold,使用内存占总内存的比列，比如0.75)时，触发垃圾回收操作。有一个对象死亡的例外，java.lang.Thread类型的对象即使没有引用，只要线程还在运行，就不会被回收。

#### 串行回收器(Serial Collector)

单线程执行回收操作，回收期间暂停所有应用线程的执行，client模式下的默认回收器，通过-XX:+UseSerialGC命令行可选项强制指定。**参数可以设置使用新生代串行和老年代串行回收器**

年轻代的回收算法(Minor Collection)  
把Eden区的存活对象移到To区，To区装不下直接移到年老代，把From区的移到To区，To区装不下直接移到年老代，From区里面年龄很大的升级到年老代。 回收结束之后，Eden和From区都为空，此时把From和To的功能互换，From变To，To变From，每一轮回收之前To都是空的。设计的选型为复制。

年老代的回收算法(Full Collection)  
年老代的回收分为三个步骤，标记(Mark)、清除(Sweep)、合并(Compact)。标记阶段把所有存活的对象标记出来，清除阶段释放所有死亡的对象，合并阶段 把所有活着的对象合并到年老代的前部分，把空闲的片段都留到后面。设计的选型为合并，减少内存的碎片。

#### 并行回收

##### 并行回收器(ParNew回收器)

并行回收器在串行回收器基础上做了改进，他可以使用多个线程同时进行垃  
圾回收，对于计算能力强的计算机而言，可以有效的缩短垃圾回收所需的尖  
际时间。  
**ParNew回收器是一个工作在新生代的垃圾收集器**，他只是简单的将串行回收  
器多线程快他的回收策略和算法和串行回收器一样。  
使用**XX:+UseParNewGC 新生代ParNew回收器**，老年代则使用市行回收器  
ParNew回收器工作时的线程数量可以使用**XX:ParaleiGCThreads**参数指  
定，一般最好和计算机的CPU相当，避免过多的栽程影响性能。

##### 并行回收集器(ParallelGC)

老年代ParallelOldGC回收器也是一种多线程的回收器，和新生代的  
ParallelGC回收器一样，也是一种关往吞吐量的回收器，他使用了标记压缩  
算法进行实现。  
-XX:+UseParallelOldGC 进行设置  
-XX:+ParallelCThread也可以设置垃圾收集时的线程教量。

#### 并CMS(并发GC)收集器

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的收集器。CMS收集器是基于“标记-清除”算法实现的，整个收集过程大致分为4个步骤：

①.初始标记(CMS initial mark)

②.并发标记(CMS concurrenr mark)

③.重新标记(CMS remark)

④.并发清除(CMS concurrent sweep)

     其中初始标记、重新标记这两个步骤任然需要停顿其他用户线程。初始标记仅仅只是标记出GC ROOTS能直接关联到的对象，速度很快，并发标记阶段是进行GC ROOTS 根搜索算法阶段，会判定对象是否存活。而重新标记阶段则是为了修正并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间会被初始标记阶段稍长，但比并发标记阶段要短。

     由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以整体来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

CMS收集器的优点：并发收集、低停顿，但是CMS还远远达不到完美，器主要有三个显著缺点：

CMS收集器对CPU资源非常敏感。在并发阶段，虽然不会导致用户线程停顿，但是会占用CPU资源而导致引用程序变慢，总吞吐量下降。CMS默认启动的回收线程数是：(CPU数量+3) / 4。

CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure“，失败后而导致另一次Full  GC的产生。由于CMS并发清理阶段用户线程还在运行，伴随程序的运行自热会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在本次收集中处理它们，只好留待下一次GC时将其清理掉。这一部分垃圾称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，  
即需要预留足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分内存空间提供并发收集时的程序运作使用。在默认设置下，CMS收集器在老年代使用了68%的空间时就会被激活，也可以通过参数-XX:CMSInitiatingOccupancyFraction的值来提供触发百分比，以降低内存回收次数提高性能。要是CMS运行期间预留的内存无法满足程序其他线程需要，就会出现“Concurrent Mode Failure”失败，这时候虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction设置的过高将会很容易导致“Concurrent Mode Failure”失败，性能反而降低。

最后一个缺点，CMS是基于“标记-清除”算法实现的收集器，使用“标记-清除”算法收集后，会产生大量碎片。空间碎片太多时，将会给对象分配带来很多麻烦，比如说大对象，内存空间找不到连续的空间来分配不得不提前触发一次Full  GC。为了解决这个问题，CMS收集器提供了一个-XX:UseCMSCompactAtFullCollection开关参数，用于在Full  GC之后增加一个碎片整理过程，还可通过-XX:CMSFullGCBeforeCompaction参数设置执行多少次不压缩的Full  GC之后，跟着来一次碎片整理过程。

#### G1回收器

G1回收器(Garbage-First)实在]dk1.7中提出的垃圾回收器，从长期目标来看是为了取  
代CMS回收器，G1回收器拥有独特的垃圾回收策略，G1属于分代垃圾回收器，区分  
新生代和老年代，依然有eden和from/to区,它并不要求整个eden区或者新生代、老  
年代的空间都连续，它使用了分区算法。  
并行性: G1回收期间可多线程同时工作。  
井发性G1拥有与应用程序交替执行能力，部分工作可与应用程序同时执行，在整个  
GC期间不会完全阻塞应用程序。  
分代GC:G1依然是一个分代的收集器，但是它是非两新生代和老年代一杯政的杂尊。  
空间基理，G1在国收过程中，不会微CMS那样在若千tacAy 要进行碎片整理。  
G1  
来用了有效复制对象的方式，减少空间碎片。  
利得程，用于分区的原因，G可以贝造取都分区城进行回收，帽小了国收的格想，  
提升了性能。  
使用.XXX:+UseG1GC 应用G1收集器，  
Mills指定最大停顿时间  
使用-XX:MaxGCPausel  
设置并行回收的线程数量  
使用-XX:ParallelGCThreads

### 调优总结

**初始堆值和最大堆内存内存越大，吞吐量就越高。**

**最好使用并行收集器,因为并行手机器速度比串行吞吐量高，速度快。**

**设置堆内存新生代的比例和老年代的比例最好为1:2或者1:3。**

**减少GC对老年代的回收。**

## MySQL优化

### MySQL如何优化

表的设计合理化(符合3NF)

添加适当索引(index) [四种: 普通索引、主键索引、唯一索引unique、全文索引]

SQL语句优化

分表技术(水平分割、垂直分割)

读写[写: update/delete/add]分离

存储过程 [模块化编程，可以提高速度]

对mysql配置优化 [配置最大并发数my.ini, 调整缓存大小 ]

mysql服务器硬件升级

定时的去清除不需要的数据,定时进行碎片整理(MyISAM)

### 数据库设计

### 什么是数据库范式

为了建立冗余较小、结构合理的数据库，设计数据库时必须遵循一定的规则。在关系型数据库中这种规则就称为范式。范式是符合某一种设计要求的总结。要想设计一个结构合理的关系型数据库，必须满足一定的范式。

### 数据库三大范式

第一范式：1NF是对属性的原子性约束，要求属性(列)具有原子性，不可再分解；(只要是关系型数据库都满足1NF)

第二范式：2NF是对记录的惟一性约束，表中的记录是唯一的, 就满足2NF, 通常我们设计一个主键来实现，主键不能包含业务逻辑。

第三范式：3NF是对字段冗余性的约束，它要求字段没有冗余。 没有冗余的数据库设计可以做到。

但是，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。具体做法是： 在概念数据模型设计时遵守第三范式，降低范式标准的工作放到物理数据模型设计时考虑。降低范式就是增加字段，允许冗余。

### 慢查询

#### 什么是慢查询

MySQL默认10秒内没有响应SQL结果,则为慢查询

可以去修改MySQL慢查询默认时间

#### 如何修改慢查询

|  |
| --- |
| --查询慢查询时间  show variables like 'long\_query\_time';  --修改慢查询时间  set long\_query\_time=1; ---但是重启mysql之后，long\_query\_time依然是my.ini中的值 |

### 如何将慢查询定位到日志中

在默认情况下，我们的mysql不会记录慢查询，需要在启动mysql时候，指定记录慢查询才可以

bin\mysqld.exe --safe-mode --slow-query-log [mysql5.5 可以在my.ini指定]（安全模式启动，数据库将操作写入日志，以备恢复）

bin\mysqld.exe –log-slow-queries=d:/abc.log [低版本mysql5.0可以在my.ini指定]

先关闭mysql,再启动, 如果启用了慢查询日志，默认把这个文件放在

my.ini 文件中记录的位置

#Path to the database root

datadir=" C:/ProgramData/MySQL/MySQL Server 5.5/Data/"

### 索引

#### 什么是索引

索引用来快速地寻找那些具有特定值的记录，所有MySQL索引都以B-树的形式保存。如果没有索引，执行查询时MySQL必须从第一个记录开始扫描整个表的所有记录，直至找到符合要求的记录。表里面的记录数量越多，这个操作的代价就越高。如果作为搜索条件的列上已经创建了索引，MySQL无需扫描任何记录即可迅速得到目标记录所在的位置。如果表有1000个记录，通过索引查找记录至少要比顺序扫描记录快100倍。

#### 索引的分类

##### 主键索引

主键是一种唯一性索引，但它必须指定为“PRIMARY KEY”。如果你曾经用过AUTO\_INCREMENT类型的列，你可能已经熟悉主键之类的概念了。主键一般在创建表的时候指定，例如“CREATE TABLE tablename ( [...], PRIMARY KEY (列的列表) ); ”。但是，我们也可以通过修改表的方式加入主键，例如“ALTER TABLE tablename ADD PRIMARY KEY (列的列表); ”。每个表只能有一个主键。

##### 创建主键索引

主键是一种唯一性索引，但它必须指定为“PRIMARY KEY”。如果你曾经用过AUTO\_INCREMENT类型的列，你可能已经熟悉主键之类的概念了。主键一般在创建表的时候指定，例如“CREATE TABLE tablename ( [...], PRIMARY KEY (列的列表) ); ”。但是，我们也可以通过修改表的方式加入主键，例如“ALTER TABLE tablename ADD PRIMARY KEY (列的列表); ”。每个表只能有一个主键。

当一张表，把某个列设为主键的时候，则该列就是主键索引

create table aaa

(id int unsigned primary key auto\_increment ,

name varchar(32) not null default '');

这是id 列就是主键索引.

create table bbb (id int , name varchar(32) not null default '');

如果你创建表时，没有指定主键索引，也可以在创建表后，在添加, 指令:

实例:

alter table 表名 add primary key (列名);

删除主键索引

alter table articles drop primary key;

##### 查询索引

desc 表名; 不能显示索引名称

show index from 表名

show keys from 表名

##### 全文索引

错误用法:

select \* from articles where body like '%mysql%'; 错误用法 索引不会生效

正确用法:

select \* from articles where match(title,body) against ( 'database')

说明:

1. 在mysql中fulltext 索引只针对 myisam生效
2. mysql自己提供的fulltext针对英文生效->sphinx (coreseek) 技术处理中文
3. 使用方法是 match(字段名..) against(‘关键字’)
4. 全文索引：停止词, 因为在一个文本中，创建索引是一个无穷大的数，因此，对一些常用词和字符，就不会创建，这些词，称为停止词.比如（a，b，mysql，the）

mysql> select match(title,body) against ('database') from articles;（输出的是每行和database的匹配度）

##### 唯一索引

这种索引和前面的“普通索引”基本相同，但有一个区别：索引列的所有值都只能出现一次，即必须唯一。唯一性索引可以用以下几种方式创建：

创建索引，例如CREATE UNIQUE INDEX <索引的名字> ON tablename (列的列表)；

修改表，例如ALTER TABLE tablename ADD UNIQUE [索引的名字] (列的列表)；

创建表的时候指定索引，例如CREATE TABLE tablename ( [...], UNIQUE [索引的名字] (列的列表) )；

##### 普通索引

 普通索引（由关键字KEY或INDEX定义的索引）的唯一任务是加快对数据的访问速度。因此，应该只为那些最经常出现在查询条件（WHEREcolumn=）或排序条件（ORDERBYcolumn）中的数据列创建索引。只要有可能，就应该选择一个数据最整齐、最紧凑的数据列（如一个整数类型的数据列）来创建索引。

create table ccc(

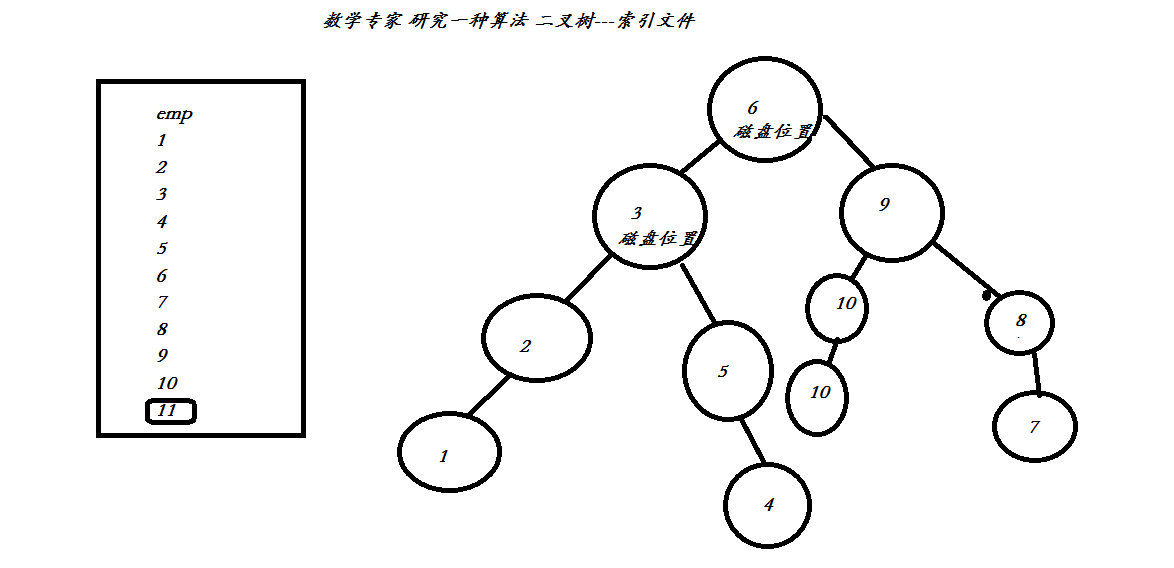
id int unsigned,

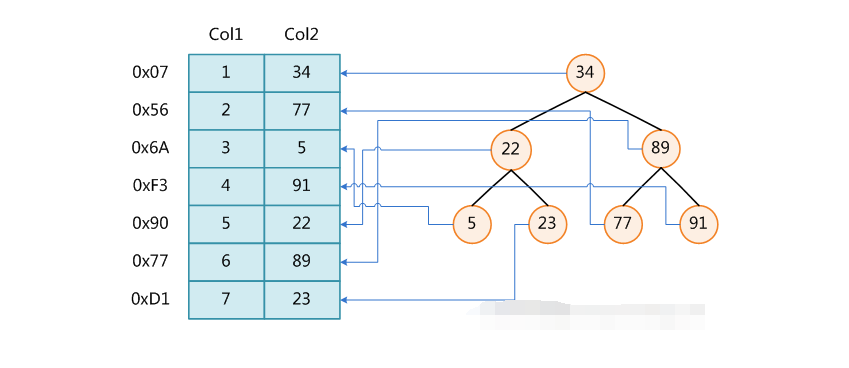
name varchar(32)

)

create index 索引名 on 表 (列1,列名2);

##### 索引的实现原理





**数据库索引**，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。**索引的实现通常使用 B 树及其变种 B+ 树**。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

上图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快 Col2 的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在 O(log2n)的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。**一般来说，应该在这些列上创建索引：**在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在 WHERE 子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。**一般来说，不应该创建索引的的这些列具有下列特点：**

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

第三，对于那些定义为 text, image 和 bit 数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在数据库设计器中创建三种索引：**唯一索引、主键索引和聚集索引**。

**唯一索引**

唯一索引是不允许其中任何两行具有相同索引值的索引。

当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在 employee 表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。**主键索引**数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。**聚集索引**在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘 I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的**局部性原理**：**当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。**

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高 I/O 效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为 4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

B-/+Tree 索引的性能分析

到这里终于可以分析 B-/+Tree 索引的性能了。

上文说过一般使用磁盘 I/O 次数评价索引结构的优劣。先从 B-Tree 分析，根据 B-Tree 的定义，可知检索一次最多需要访问 h 个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全载入。为了达到这个目的，在实际实现 B-Tree 还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个 node 只需一次 I/O。

**B-Tree 中一次检索最多需要 h-1 次 I/O（根节点常驻内存），渐进复杂度为 O(h)=O(logdN)。**一般实际应用中，出度 d 是非常大的数字，通常超过 100，因此 h 非常小（通常不超过 3）。

而红黑树这种结构，h 明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的 I/O 渐进复杂度也为 O(h)，效率明显比 B-Tree 差很多。

**综上所述，用 B-Tree 作为索引结构效率是非常高的。**

**应该花时间学习 B-树和 B+ 树数据结构**

=============================================================================================================

1）B 树

B 树中每个节点包含了键值和键值对于的数据对象存放地址指针，所以成功搜索一个对象可以不用到达树的叶节点。

成功搜索包括节点内搜索和沿某一路径的搜索，成功搜索时间取决于关键码所在的层次以及节点内关键码的数量。

在 B 树中查找给定关键字的方法是：首先把根结点取来，在根结点所包含的关键字 K1,…,kj 查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查的关键字在某个 Ki 或 Ki+1 之间，于是取 Pi 所指的下一层索引节点块继续查找，直到找到，或指针 Pi 为空时查找失败。

2）B+ 树

B+ 树非叶节点中存放的关键码并不指示数据对象的地址指针，非也节点只是索引部分。所有的叶节点在同一层上，包含了全部关键码和相应数据对象的存放地址指针，且叶节点按关键码从小到大顺序链接。如果实际数据对象按加入的顺序存储而不是按关键码次数存储的话，叶节点的索引必须是稠密索引，若实际数据存储按关键码次序存放的话，叶节点索引时稀疏索引。

B+ 树有 2 个头指针，一个是树的根节点，一个是最小关键码的叶节点。

所以 B+ 树有两种搜索方法：

一种是按叶节点自己拉起的链表顺序搜索。

一种是从根节点开始搜索，和 B 树类似，不过如果非叶节点的关键码等于给定值，搜索并不停止，而是继续沿右指针，一直查到叶节点上的关键码。所以无论搜索是否成功，都将走完树的所有层。

B+ 树中，数据对象的插入和删除仅在叶节点上进行。

这两种处理索引的数据结构的不同之处：  
a，B 树中同一键值不会出现多次，并且它有可能出现在叶结点，也有可能出现在非叶结点中。而 B+ 树的键一定会出现在叶结点中，并且有可能在非叶结点中也有可能重复出现，以维持 B+ 树的平衡。  
b，因为 B 树键位置不定，且在整个树结构中只出现一次，虽然可以节省存储空间，但使得在插入、删除操作复杂度明显增加。B+ 树相比来说是一种较好的折中。  
c，B 树的查询效率与键在树中的位置有关，最大时间复杂度与 B+ 树相同(在叶结点的时候)，最小时间复杂度为 1(在根结点的时候)。而 B+ 树的时候复杂度对某建成的树是固定的。可以扫描2的次方。

##### 索引的代价

占用磁盘空间

对DML(update、delete、insert)语句的效率影响

增删改会对索引影响，因为索引要重新整理。

|  |  |
| --- | --- |
| 存储引擎 | 允许的索引类型 |
| myisam | btree |
| innodb | btree |
| memory/yeap | Hash,btree |

##### 那些列上适合添加索引

1. 查询作为查询条件字段应该创建索引
2. 唯一性太差的字段不适合单独创建索引,即使频繁

Select \* from emp where sex=’男’

1. 频繁更新字段，也不要定义索引。
2. 不会出现在where语句的字段不要创建索引

总结:满处一下条件的字段，才应该创建索引

1. 肯定在where条件经常使用
2. 该字段的内容不是唯一的几个值
3. 字段内容不是频繁变化

##### 索引的注意事项

创建主键索引

alter table 表名 add primary key (列名);

创建一个联合索引

alter table dept add index my\_ind (dname,loc); // dname 左边的列,loc就是右边的列

注意:

1.对于创建的多列索引，如果不是使用第一部分，则不会创建索引。

explain select \* from dept where loc='aaa'\G

就不会使用到索引

2.模糊查询在like前面有百分号开头会失效。

3. 如果条件中有or，即使其中有条件带索引也不会使用。换言之，就是要求使用的所有字段，都必须建立索引, 我们建议大家尽量避免使用or 关键字

4.如果列类型是字符串，那一定要在条件中将数据使用引号引用起来。否则不使用索引。(添加时,字符串必须’’), 也就是，如果列是字符串类型，就一定要用 ‘’ 把他包括起来.

5.如果mysql估计使用全表扫描要比使用索引快，则不使用索引。

##### 查询所用使用率

show status like ‘handler\_read%’;

大家可以注意：  
handler\_read\_key:这个值越高越好，越高表示使用索引查询到的次数。

handler\_read\_rnd\_next:这个值越高，说明查询低效。

### SQL调优

1. 使用group by 分组查询是，默认分组后，还会排序，可能会降低速度，

在group by 后面增加 order by null 就可以防止排序.

explain select \* from emp group by deptno order by null;

1. 有些情况下，可以使用连接来替代子查询。因为使用join，MySQL不需要在内存中创建临时表。

select \* from dept, emp where dept.deptno=emp.deptno; [简单处理方式]

select \* from dept left join emp on dept.deptno=emp.deptno; [左外连接，更ok!]

1. 对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引

应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

最好不要给数据库留 NULL，尽可能的使用 NOT NULL 填充数据库.

备注、描述、评论之类的可以设置为 NULL，其他的，最好不要使用 NULL。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL 也包含在内），都是占用 100 个字符的空间的，如果是 varchar 这样的变长字段， null 不占用空间。

可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：

select id from t where num = 0

更多mysql sql语句调优查看http://bbs.itmayiedu.com/article/1511164574773

### MySQL数据引擎

使用的存储引擎 myisam / innodb/ memory

myisam 存储: 如果表对事务要求不高，同时是以查询和添加为主的，我们考虑使用myisam存储引擎. ,比如 bbs 中的 发帖表，回复表.

INNODB 存储: 对事务要求高，保存的数据都是重要数据，我们建议使用INNODB,比如订单表，账号表.

MyISAM 和 INNODB的区别

1. 事务安全（MyISAM不支持事务，INNODB支持事务）

2. 查询和添加速度（MyISAM批量插入速度快）

3. 支持全文索引（MyISAM支持全文索引，INNODB不支持全文索引）

4. 锁机制（MyISAM时表锁，innodb是行锁）

5. 外键 MyISAM 不支持外键， INNODB支持外键. (在PHP开发中，通常不设置外键，通常是在程序中保证数据的一致)

Memory 存储，比如我们数据变化频繁，不需要入库，同时又频繁的查询和修改，我们考虑使用memory, 速度极快. （如果mysql重启的话，数据就不存在了）



#### Myisam注意事项

如果你的数据库的存储引擎是myisam,请一定记住要定时进行碎片整理

举例说明:

create table test100(id int unsigned ,name varchar(32))engine=myisam;

insert into test100 values(1,’aaaaa’);

insert into test100 values(2,’bbbb’);

insert into test100 values(3,’ccccc’);

我们应该定义对myisam进行整理

optimize table test100;

### 数据库数据备份

#### 手动方式

cmd控制台:

在环境变量中配置mysql环境变量

mysqldump –u -账号 –密码 数据库 [表名1 表名2..] > 文件路径

案例: mysqldump -u -root root test > d:\temp.sql

比如: 把temp数据库备份到 d:\temp.bak

mysqldump -u root -proot test > f:\temp.bak

如果你希望备份是，数据库的某几张表

mysqldump -u root -proot test dept > f:\temp.dept.sql

如何使用备份文件恢复我们的数据.

mysql控制台

source d:\temp.dept.bak

#### 自动方式

把备份数据库的指令，写入到 bat文件, 然后通过任务管理器去定时调用 bat文件.

mytask.bat 内容是:

|  |
| --- |
| @echo off  F:\path\mysqlanzhuang\bin\mysqldump -u root -proot test dept > f:\temp.dept.sql |

创建执行计划任务执行脚本。

### 分表分库

#### 垂直拆分

垂直拆分就是要把表按模块划分到不同**数据库**表中（当然原则还是不破坏第三范式），这种拆分在大型网站的演变过程中是很常见的。当一个网站还在很小的时候，只有小量的人来开发和维护，各模块和表都在一起，当网站不断丰富和壮大的时候，也会变成多个子系统来支撑，这时就有按模块和功能把表划分出来的需求。其实，相对于垂直切分更进一步的是服务化改造，说得简单就是要把原来强耦合的系统拆分成多个弱耦合的服务，通过服务间的调用来满足业务需求看，因此表拆出来后要通过服务的形式暴露出去，而不是直接调用不同模块的表，淘宝在架构不断演变过程，最重要的一环就是服务化改造，把用户、交易、店铺、宝贝这些核心的概念抽取成独立的服务，也非常有利于进行局部的优化和治理，保障核心模块的稳定性

垂直拆分用于分布式场景。

#### 水平拆分

上面谈到垂直切分只是把表按模块划分到不同数据库，但没有解决单表大数据量的问题，而水平切分就是要把一个表按照某种规则把数据划分到不同表或数据库里。例如像计费系统，通过按时间来划分表就比较合适，因为系统都是处理某一时间段的数据。而像SaaS应用，通过按用户维度来划分数据比较合适，因为用户与用户之间的隔离的，一般不存在处理多个用户数据的情况，简单的按user\_id范围来水平切分

通俗理解：水平拆分行，行数据拆分到不同表中， 垂直拆分列，表数据拆分到不同表中

#### 水平分割案例

思路:在大型电商系统中，每天的会员人数不断的增加。达到一定瓶颈后如何优化查询。

可能大家会想到索引，万一用户量达到上亿级别，如何进行优化呢？

使用水平分割拆分数据库表。

#### 如何使用水平拆分数据库

使用水平分割拆分表，具体根据业务需求，有的按照注册时间、取摸、账号规则、年份等。

#### 使用取摸方式分表

首先我创建三张表 user0 / user1 /user2 , 然后我再创建 uuid表，该表的作用就是提供自增的id。

|  |
| --- |
| create table user0(  id int unsigned primary key ,  name varchar(32) not null default '',  pwd varchar(32) not null default '')  engine=myisam charset utf8;  create table user1(  id int unsigned primary key ,  name varchar(32) not null default '',  pwd varchar(32) not null default '')  engine=myisam charset utf8;  create table user2(  id int unsigned primary key ,  name varchar(32) not null default '',  pwd varchar(32) not null default '')  engine=myisam charset utf8;  create table uuid(  id int unsigned primary key auto\_increment)engine=myisam charset utf8; |

##### 创建一个demo项目

#### POM文件

|  |
| --- |
| **<parent>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-parent</artifactId>**  **<version>1.3.3.RELEASE</version>**  **</parent>**  **<dependencies>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-jdbc</artifactId>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter</artifactId>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-test</artifactId>**  **<scope>test</scope>**  **</dependency>**  **<dependency>**  **<groupId>mysql</groupId>**  **<artifactId>mysql-connector-java</artifactId>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-web</artifactId>**  **</dependency>**  **</dependencies>** |

#### Service代码

|  |
| --- |
| **@Service**  **public class UserService {**  **@Autowired**  **private JdbcTemplate jdbcTemplate;**  **public String regit(String name, String pwd) {**  **// 1.先获取到 自定增长ID**  **String idInsertSQL = "INSERT INTO uuid VALUES (NULL);";**  **jdbcTemplate.update(idInsertSQL);**  **Long insertId = jdbcTemplate.queryForObject("select last\_insert\_id()", Long.class);**  **// 2.判断存储表名称**  **String tableName = "user" + insertId % 3;**  **// 3.注册数据**  **String insertUserSql = "INSERT INTO " + tableName + " VALUES ('" + insertId + "','" + name + "','" + pwd**  **+ "');";**  **System.*out*.println("insertUserSql:" + insertUserSql);**  **jdbcTemplate.update(insertUserSql);**  **return "success";**  **}**  **public String get(Long id) {**  **String tableName = "user" + id % 3;**  **String sql = "select name from " + tableName + " where id="+id;**  **System.*out*.println("SQL:" + sql);**  **String name = jdbcTemplate.queryForObject(sql, String.class);**  **return name;**  **}**  **}** |

#### Controller

|  |
| --- |
| **@RestController**  **public class UserController {**  **@Autowired**  **private UserService userService;**  **@RequestMapping("/regit")**  **public String regit(String name, String pwd) {**  **return userService.regit(name, pwd);**  **}**  **@RequestMapping("/get")**  **public String get(Long id) {**  **String name = userService.get(id);**  **return name;**  **}**  **}** |

### 什么是读写分离

在数据库集群架构中，让主库负责处理事务性查询，而从库只负责处理select查询，让两者分工明确达到提高数据库整体读写性能。当然，主数据库另外一个功能就是负责将事务性查询导致的数据变更同步到从库中，也就是写操作。

#### 读写分离的好处

 1）分摊服务器压力，提高机器的系统处理效率

    读写分离适用于读远比写的场景，如果有一台服务器，当select很多时，update和delete会被这些select访问中的数据堵塞，等待select结束，并发性能并不高，而主从只负责各自的写和读，极大程度的缓解X锁和S锁争用；

   假如我们有1主3从，不考虑上述1中提到的从库单方面设置，假设现在1分钟内有10条写入，150条读取。那么，1主3从相当于共计40条写入，而读取总数没变，因此平均下来每台服务器承担了10条写入和50条读取（主库不承担读取操作）。因此，虽然写入没变，但是读取大大分摊了，提高了系统性能。另外，当读取被分摊后，又间接提高了写入的性能。所以，总体性能提高了，说白了就是拿机器和带宽换性能；

  2）增加冗余，提高服务可用性，当一台数据库服务器宕机后可以调整另外一台从库以最快速度恢复服务

#### 主从复制原理

依赖于二进制日志，binary-log.

二进制日志中记录引起数据库发生改变的语句

Insert 、delete、update、create table

#### Scale-up与Scale-out区别

Scale Out是指Application可以在水平方向上扩展。一般对数据中心的应用而言，Scale out指的是当添加更多的机器时，应用仍然可以很好的利用这些机器的资源来提升自己的效率从而达到很好的扩展性。

Scale Up是指Application可以在垂直方向上扩展。一般对单台机器而言，Scale Up值得是当某个计算节点（机器）添加更多的CPU Cores，存储设备，使用更大的内存时，应用可以很充分的利用这些资源来提升自己的效率从而达到很好的扩展性。

#### 解决问题

数据如何不被丢失

备份

读写分离

数据库负载均衡

高可用

#### 环境搭建

1. 准备环境

两台windows操作系统 ip分别为: 172.27.185.1(主)、172.27.185.2(从)

1. 连接到主服务(172.27.185.1)服务器上，给从节点分配账号权限。

GRANT REPLICATION SLAVE ON \*.\* TO 'root'@'172.27.185.2' IDENTIFIED BY 'root';

1. 在主服务my.ini文件新增

|  |
| --- |
| server-id=200  log-bin=mysql-bin  relay-log=relay-bin  relay-log-index=relay-bin-index |

重启mysql服务

1. 在从服务my.ini文件新增

|  |
| --- |
| server-id = 210  replicate-do-db =itmayiedu #需要同步数据库 |

重启mysql服务

1. 从服务同步主数据库

|  |
| --- |
| stop slave;  change  master to master\_host='172.27.185.1',master\_user='root',master\_password='root';  start slave;  show slave status; |

### MyCat

#### 什么是  Mycat

是一个开源的分布式数据库系统，但是因为数据库一般都有自己的数据库引擎，而Mycat并没有属于自己的独有数据库引擎，所有严格意义上说并不能算是一个完整的数据库系统，只能说是一个在应用和数据库之间起桥梁作用的中间件。

在Mycat中间件出现之前，MySQL主从复制集群，如果要实现读写分离，一般是在程序段实现，这样就带来了一个问题，即数据段和程序的耦合度太高，如果数据库的地址发生了改变，那么我的程序也要进行相应的修改，如果数据库不小心挂掉了，则同时也意味着程序的不可用，而对于很多应用来说，并不能接受；

  引入Mycat中间件能很好地对程序和数据库进行解耦，这样，程序只需关注数据库中间件的地址，而无需知晓底层数据库是如何提供服务的，大量的通用数据聚合、事务、数据源切换等工作都由中间件来处理；

  Mycat中间件的原理是对数据进行分片处理，从原有的一个库，被切分为多个分片数据库，所有的分片数据库集群构成完成的数据库存储，有点类似磁盘阵列中的RAID0.

# JavaEE

## 基础部分

### BS与CS区别？

**C/S (Client - Server 客户端-服务器端)**

**典型应用：QQ软件 ，飞秋，红蜘蛛。**

**特点：**

**1）必须下载特定的客户端程序。**

**2）服务器端升级，客户端升级。**

**B/S （Broswer -Server 浏览器端- 服务器端）**

**典型应用： 腾讯官方（www.qq.com） 163新闻网站， 蚂蚁课堂官网（俗称：网站）**

**特点：**

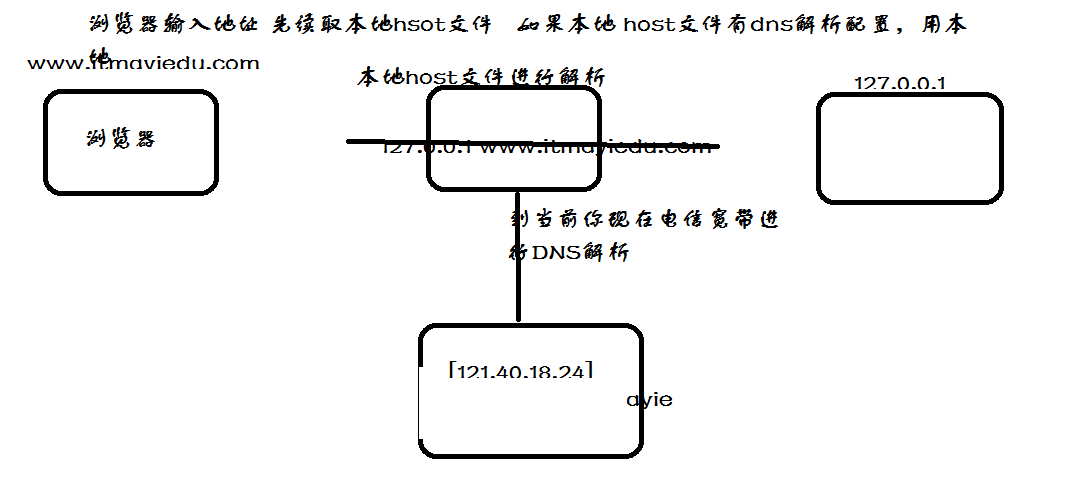
**1）不需要安装特定的客户端（只需要安装浏览器即可！！）**

**2）服务器端升级，浏览器不需要升级！！！！**

**javaweb的程序就是b/s软件结构！！！**

### DNS解析过程

**1、在浏览器中输入www.qq.com域名，操作系统会先检查自己本地的hosts文件是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。   
2、如果hosts里没有这个域名的映射，则查找本地DNS解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。**



### 外网映射工具

**外网映射工具的作用,主要将本地服务映射到外网。**

**应用场景:支付回调、微信开发、对接第三方接口等。**

**映射工具Ngrok、花生壳等。**

### 静态资源和动态资源的区别

**静态资源： 当用户多次访问这个资源，资源的源代码永远不会改变的资源。**

**动态资源：当用户多次访问这个资源，资源的源代码可能会发送改变。**

### Sevlet的生命周期（重点）

**构造方法： 创建servlet对象的时候调用。**

**默认情况下，第一次访问servlet的时候创建servlet对象 只调用1次。证明servlet对象在tomcat是单实例的。**

**init方法： 创建完servlet对象的时候调用。只调用1次。**

**service方法： 每次发出请求时调用。调用n次。**

**destroy方法： 销毁servlet对象的时候调用。停止服务器或者重新部署web应用时销毁servlet对象。只调用1次。**

### 怎么证明Servlet是单例的？

**因为Servlet是通过Java反射机制，读取web.xml配置中的servlet-class 完整路径，进行反射默认执行无参构造函数，所以只要servlet类执行无参构造函数永远只执行一遍，则Servlet是单例的。**

### Servlet的多线程并发问题

**注意： servlet对象在tomcat服务器是单实例多线程的。**

**因为servlet是多线程的，所以当多个servlet的线程同时访问了servlet的共享数据，如成员变量，可能会引发线程安全问题。**

**解决办法：**

**1）把使用到共享数据的代码块进行同步（使用synchronized关键字进行同步）**

**2）建议在servlet类中尽量不要使用成员变量。如果确实要使用成员，必须同步。而且尽量缩小同步代码块的范围。（哪里使用到了成员变量，就同步哪里！！），以避免因为同步而导致并发效率降低。**

### 转发与重定向区别？

**1）转发**

**a）地址栏不会改变**

**b）转发只能转发到当前web应用内的资源**

**c）可以在转发过程中，可以把数据保存到request域对象中**

**2）重定向**

**a）地址栏会改变，变成重定向到地址。**

**b）重定向可以跳转到当前web应用，或其他web应用，甚至是外部域名网站。**

**c）不能再重定向的过程，把数据保存到request中。**

### 重定向实现原理

**重定向会发送两次请求,浏览器认识状态码为302，会再次向服务器发送一次请求，获取请求头的中location的value值进行重定向。**

### JavaWeb有哪些会话技术

**Cookie会话数据保存在浏览器客户端**

**服务器创建Cookie,将Cookie内容以响应头方式发送给客户端存放在本地,当下次发送请求时.会将Cookie信息以请求方式发送给服务器端**

**注意:Cookie信息不能夸浏览器访问**

**Session会话保存与服务器端**

**服务器创建Session,Session内容存放服务器端上,以响应头方式将SessionId发送给客户端保存，当下次发送请求时,会将SessionID 以请求头方式发送给服务器端**

**注意: 浏览器关闭,只是清除了Sessionid，并没有清除Session**

### Cookie的实现原理

1）服务器创建cookie对象，把会话数据存储到cookie对象中。

new Cookie("name","value");

2） 服务器发送cookie信息到浏览器

response.addCookie(cookie);

举例： set-cookie: name=eric (隐藏发送了一个set-cookie名称的响应头)

3）浏览器得到服务器发送的cookie，然后保存在浏览器端。

4）浏览器在下次访问服务器时，会带着cookie信息

举例： cookie: name=eric (隐藏带着一个叫cookie名称的请求头)

5）服务器接收到浏览器带来的cookie信息

request.getCookies();

### Cookie的应用场景

**购物车、显示用户上次访问的时间**

### Session的实现原理

**1）第一次访问创建session对象，给session对象分配一个唯一的ID，叫JSESSIONID**

**new HttpSession();**

**2）把JSESSIONID作为Cookie的值发送给浏览器保存**

**Cookie cookie = new Cookie("JSESSIONID", sessionID);**

**response.addCookie(cookie);**

**3）第二次访问的时候，浏览器带着JSESSIONID的cookie访问服务器**

**4）服务器得到JSESSIONID，在服务器的内存中搜索是否存放对应编号的session对象。**

**if(找到){**

**return map.get(sessionID);**

**}**

**Map<String,HttpSession>]**

**<"s001", s1>**

**<"s001,"s2>**

**5）如果找到对应编号的session对象，直接返回该对象**

**6）如果找不到对应编号的session对象，创建新的session对象，继续走1的流程**

**结论：通过JSESSION的cookie值在服务器找session对象！！！！！**

### 什么是token

**token其实就是一个令牌,具有随机性,类似于sessionId。**

**在对接一些第三方平台的时候,为了能够保证数据安全性,通常会使用一些令牌进行交互**

**例如: https://mp.weixin.qq.com/wiki?t=resource/res\_main&id=mp1421140183**

### 如何自定义token

**token生成规则,只要保证token生成一个不重复的唯一字符串即可。**

**使用jdk自带的uuid生成规则。**

### 什么是UUID

**UUID含义是通用唯一识别码 (Universally Unique Identifier)，这是一个软件建构的标准，也是被开源软件基金会 (Open Software Foundation, OSF)   
的组织应用在分布式计算环境 (Distributed Computing Environment, DCE) 领域的一部分。**

**UUID 的目的，是让分布式系统中的所有元素，都能有唯一的辨识资讯，而不需要透过中央控制端来做辨识资讯的指定。如此一来，每个人都可以建立不与其它人冲突的 UUID。  
在这样的情况下，就不需考虑数据库建立时的名称重复问题。目前最广泛应用的 UUID，即是微软的 Microsoft's Globally Unique Identifiers (GUIDs)，而其他重要的应用，  
则有 Linux ext2/ext3 档案系统、LUKS 加密分割区、GNOME、KDE、Mac OS X 等等**

### UUID组成

**UUID保证对在同一时空中的所有机器都是唯一的。通常平台会提供生成的API。按照开放软件基金会(OSF)制定的标准计算，用到了以太网卡地址、纳秒级时间、芯片ID码和许多可能的数字  
UUID由以下几部分的组合：  
（1）当前日期和时间，UUID的第一个部分与时间有关，如果你在生成一个UUID之后，过几秒又生成一个UUID，则第一个部分不同，其余相同。  
（2）时钟序列。  
（3）全局唯一的IEEE机器识别号，如果有网卡，从网卡MAC地址获得，没有网卡以其他方式获得。  
UUID的唯一缺陷在于生成的结果串会比较长。关于UUID这个标准使用最普遍的是微软的GUID(Globals Unique Identifiers)。在ColdFusion中可以用CreateUUID()函数很简单地生成UUID，  
其格式为：xxxxxxxx-xxxx- xxxx-xxxxxxxxxxxxxxxx(8-4-4-16)，其中每个 x 是 0-9 或 a-f 范围内的一个十六进制的数字。而标准的UUID格式为：xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx (8-4-4-4-12);**

### UUID代码

|  |
| --- |
| **UUID.randomUUID().toString()** |

### 什么是Filter

**Filter 是Servlet技术中的过滤器,主要做拦截请求作用，一般用于防御XSS攻击、权限、登录判断等。**

## 网络通讯

### http部分

#### 什么是http协议

**http协议： 对浏览器客户端 和 服务器端 之间数据传输的格式规范**

#### Http格式的分类

**请求行**

**请求头**

**请求内容**

**响应行**

**响应头**

**响应内容**

#### https与http区别

**虽然说 HTTPS 有很大的优势，但其相对来说，还是存在不足之处的：**

**（1）HTTPS 协议握手阶段比较费时，会使页面的加载时间延长近 50%，增加 10% 到 20% 的耗电；**

**（2）HTTPS 连接缓存不如 HTTP 高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；**

**（3）SSL 证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。**

**（4）SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名，IPv4 资源不可能支撑这个消耗。**

**（5）HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。**

#### https请求方式

**常见的请求方式： GET 、 POST、 HEAD、 TRACE、 PUT、 CONNECT 、DELETE**

**常用的请求方式： GET 和 POST**

**表单提交：**

**<form action="提交地址" method="GET/POST">**

**<form>**

**GET vs POST 区别**

**1）GET方式提交**

**a）地址栏（URI）会跟上参数数据。以？开头，多个参数之间以&分割。**

|  |
| --- |
| **GET /day09/testMethod.html?name=eric&password=123456 HTTP/1.1**  **Host: localhost:8080**  **User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0**  **Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8**  **Accept-Language: zh-cn,en-us;q=0.8,zh;q=0.5,en;q=0.3**  **Accept-Encoding: gzip, deflate**  **Referer: http://localhost:8080/day09/testMethod.html**  **Connection: keep-alive** |

**b）GET提交参数数据有限制，不超过1KB。**

**c）GET方式不适合提交敏感密码。**

**d）注意： 浏览器直接访问的请求，默认提交方式是GET方式**

**2）POST方式提交**

**a）参数不会跟着URI后面。参数而是跟在请求的实体内容中。没有？开头，多个参数之间以&分割。**

|  |
| --- |
| **POST /day09/testMethod.html HTTP/1.1**  **Host: localhost:8080**  **User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0**  **Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8**  **Accept-Language: zh-cn,en-us;q=0.8,zh;q=0.5,en;q=0.3**  **Accept-Encoding: gzip, deflate**  **Referer: http://localhost:8080/day09/testMethod.html**  **Connection: keep-alive**  **name=eric&password=123456** |

**b）POST提交的参数数据没有限制。**

**c）POST方式提交敏感数据。**

**3.2 请求头**

|  |
| --- |
| **Accept: text/html,image/\* -- 浏览器接受的数据类型**  **Accept-Charset: ISO-8859-1 -- 浏览器接受的编码格式**  **Accept-Encoding: gzip,compress --浏览器接受的数据压缩格式**  **Accept-Language: en-us,zh- --浏览器接受的语言**  **Host: www.it315.org:80 --（必须的）当前请求访问的目标地址（主机:端口）**  **If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT --浏览器最后的缓存时间**  **Referer: http://www.it315.org/index.jsp -- 当前请求来自于哪里**  **User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0) --浏览器类型**  **Cookie:name=eric -- 浏览器保存的cookie信息**  **Connection: close/Keep-Alive -- 浏览器跟服务器连接状态。close: 连接关闭 keep-alive：保存连接。**  **Date: Tue, 11 Jul 2000 18:23:51 GMT -- 请求发出的时间** |

#### 客户端模拟http请求工具

Postmen(谷歌插件)、RestClient

#### 服务器模拟http请求工具

httpclient、HttpURLConnection

#### 前端ajax请求

|  |
| --- |
| $.ajax({  type : 'post',  dataType : "text",  url : "http://a.a.com/a/FromUserServlet",  data : "userName=余胜军&userAge=19",  success : **function**(msg) {  alert(msg);  }  }); |

## 框架部分

### Spring

#### Spring概述

Spring框架，可以解决对象创建以及对象之间依赖关系的一种框架。

且可以和其他框架一起使用；Spring与Struts, Spring与hibernate

(起到整合（粘合）作用的一个框架)

Spring提供了一站式解决方案：

1） Spring Core spring的核心功能： IOC容器, 解决对象创建及依赖关系

2） Spring Web Spring对web模块的支持。

-🡪 可以与struts整合,让struts的action创建交给spring

-🡪 spring mvc模式

3） Spring DAO Spring 对jdbc操作的支持 【JdbcTemplate模板工具类】

4） Spring ORM spring对orm的支持：

🡪 既可以与hibernate整合，【session】

🡪 也可以使用spring的对hibernate操作的封装

5）Spring AOP 切面编程

6）SpringEE spring 对javaEE其他模块的支持

#### SpringBeanId重复会怎么样？

会报错。

#### 什么是SpringIOC

SpringIOC就是把每个bean与bean之间的关系交给第三方容器进行管理,这个容器就是Spring。

什么是AOP

#### 什么是SpringAOP

**Aop，** aspect object programming 面向切面编程

功能： 让关注点代码与业务代码分离！

**关注点,**

重复代码就叫做关注点；

**切面，**

关注点形成的类，就叫切面(类)！

面向切面编程，就是指 对很多功能都有的重复的代码抽取，再在运行的时候网业务方法上动态植入“切面类代码”。

**切入点，**

执行目标对象方法，动态植入切面代码。

可以通过切入点表达式，指定拦截哪些类的哪些方法； 给指定的类在运行的时候植入切面类代码。

**应用场景:事物、日志、权限控制**

#### SpringAOP创建方式

##### 注解方式实现AOP编程

步骤：

1） 先引入aop相关jar文件 （aspectj aop优秀组件）

spring-aop-3.2.5.RELEASE.jar 【spring3.2源码】

aopalliance.jar 【spring2.5源码/lib/aopalliance】

aspectjweaver.jar 【spring2.5源码/lib/aspectj】或【aspectj-1.8.2\lib】

aspectjrt.jar 【spring2.5源码/lib/aspectj】或【aspectj-1.8.2\lib】

注意： 用到spring2.5版本的jar文件，如果用jdk1.7可能会有问题。

需要升级aspectj组件，即使用aspectj-1.8.2版本中提供jar文件提供。

2） bean.xml中引入aop名称空间

3） 开启aop注解

4) 使用注解

@Aspect 指定一个类为切面类

@Pointcut("execution(\* com.itmayiedu.service.UserService.add(..))") 指定切入点表达式

@Before("pointCut\_()") 前置通知: 目标方法之前执行

@After("pointCut\_()") 后置通知：目标方法之后执行（始终执行）

@AfterReturning("pointCut\_()") 返回后通知： 执行方法结束前执行(异常不执行)

@AfterThrowing("pointCut\_()") 异常通知: 出现异常时候执行

@Around("pointCut\_()") 环绕通知： 环绕目标方法执行

|  |
| --- |
| **@Component**  **@Aspect**  **public class Aop {**  **@Before("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void begin() {**  **System.*out*.println("前置通知");**  **}**  **@After("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void commit() {**  **System.*out*.println("后置通知");**  **}**  **@AfterReturning("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void afterReturning() {**  **System.*out*.println("运行通知");**  **}**  **@AfterThrowing("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void afterThrowing() {**  **System.*out*.println("异常通知");**  **}**  **@Around("execution(\* com.itmayiedu.service.UserService.add(..))")**  **public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {**  **System.*out*.println("我是环绕通知-前");**  **proceedingJoinPoint.proceed();**  **System.*out*.println("我是环绕通知-后");**  **}**  **}** |

##### XML方式实现AOP编程

Xml实现aop编程：

1） 引入jar文件 【aop 相关jar， 4个】

2） 引入aop名称空间

3）aop 配置

\* 配置切面类 （重复执行代码形成的类）

\* aop配置

拦截哪些方法 / 拦截到方法后应用通知代码

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd"*>    <!-- dao 实例 -->  <bean id=*"userDao"* class=*"com.itmayiedu.UserDao"*></bean>  <bean id=*"orderDao"* class=*"com.itmayiedu.OrderDao"*></bean>    <!-- 切面类 -->  <bean id=*"aop"* class=*"com.itmayiedu.Aop"*></bean>    <!-- Aop配置 -->  <aop:config>  <!-- 定义一个切入点表达式： 拦截哪些方法 -->  <aop:pointcut expression=*"execution(\* com.itmayiedu.\*.\*(..))"* id=*"pt"*/>  <!-- 切面 -->  <aop:aspect ref=*"aop"*>  <!-- 环绕通知 -->  <aop:around method=*"around"* pointcut-ref=*"pt"*/>  <!-- 前置通知： 在目标方法调用前执行 -->  <aop:before method=*"begin"* pointcut-ref=*"pt"*/>  <!-- 后置通知： -->  <aop:after method=*"after"* pointcut-ref=*"pt"*/>  <!-- 返回后通知 -->  <aop:after-returning method=*"afterReturning"* pointcut-ref=*"pt"*/>  <!-- 异常通知 -->  <aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"pt"*/>    </aop:aspect>  </aop:config>  </beans> |

#### Spring是单例还是多例？

Spring默认是单例

#### Spring作用域

##### singleton

当一个bean的 作用域设置为singleton, 那么Spring IOC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。换言之，当把 一个bean定义设置为singleton作用域时，Spring IOC容器只会创建该bean定义的唯一实例。这个单一实例会被存储到单例缓存（singleton cache）中，并且所有针对该bean的后续请求和引用都 将返回被缓存的对象实例，这里要注意的是singleton作用域和GOF设计模式中的单例是完全不同的，单例设计模式表示一个ClassLoader中 只有一个class存在，而这里的singleton则表示一个容器对应一个bean，也就是说当一个bean被标识为singleton时 候，spring的IOC容器中只会存在一个该bean。

##### prototype

prototype作用域部署的bean，每一次请求（将其注入到另一个bean中，或者以程序的方式调用容器的 getBean()方法）都会产生一个新的bean实例，相当与一个new的操作，对于prototype作用域的bean，有一点非常重要，那就是Spring不能对一个prototype bean的整个生命周期负责，容器在初始化、配置、装饰或者是装配完一个prototype实例后，将它交给客户端，随后就对该prototype实例不闻不问了。不管何种作用域，容器都会调用所有对象的初始化生命周期回调方法，而对prototype而言，任何配置好的析构生命周期回调方法都将不会被调用。 清除prototype作用域的对象并释放任何prototype bean所持有的昂贵资源，都是客户端代码的职责。（让Spring容器释放被singleton作用域bean占用资源的一种可行方式是，通过使用 bean的后置处理器，该处理器持有要被清除的bean的引用。）

##### request

request表示该针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前HTTP request内有效，配置实例：

request、session、global session使用的时候首先要在初始化web的web.xml中做如下配置：

##### session

session作用域表示该针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前HTTP session内有效

#### IOC容器创建对象

创建对象, 有几种方式：

1） 调用无参数构造器

2） 带参数构造器

3） 工厂创建对象

工厂类，静态方法创建对象

工厂类，非静态方法创建对象

#### 依赖注入有哪些方式

Spring中，如何给对象的属性赋值? 【DI, 依赖注入】

1) 通过构造函数

2) 通过set方法给属性注入值

3) p名称空间

4) 注解

#### 事物的概述

##### ⑴ 原子性（Atomicity）

　　原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

##### ⑵ 一致性（Consistency）

　　一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

##### ⑶ 隔离性（Isolation）

　　隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

　　即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

　　关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

##### ⑷ 持久性（Durability）

　　持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

#### Spring事物控制

**编程式事务控制**

自己手动控制事务，就叫做编程式事务控制。

Jdbc代码：

Conn.setAutoCommite(false); // 设置手动控制事务

Hibernate代码：

Session.beginTransaction(); // 开启一个事务

【细粒度的事务控制： 可以对指定的方法、指定的方法的某几行添加事务控制】

(比较灵活，但开发起来比较繁琐： 每次都要开启、提交、回滚.)

**声明式事务控制**

Spring提供了对事务的管理, 这个就叫声明式事务管理。

Spring声明事物有xml方式和注解方式

Spring提供了对事务控制的实现。用户如果想用Spring的声明式事务管理，只需要在配置文件中配置即可； 不想使用时直接移除配置。这个实现了对事务控制的最大程度的解耦。

Spring声明式事务管理，核心实现就是基于Aop。

【粗粒度的事务控制： 只能给整个方法应用事务，不可以对方法的某几行应用事务。】

(因为aop拦截的是方法。)

Spring声明式事务管理器类：

Jdbc技术：**DataSourceTransactionManager**

Hibernate技术：**HibernateTransactionManager**

#### Spring事物传播行为

Spring中事务的定义：

Propagation（key属性确定代理应该给哪个方法增加事务行为。这样的属性最重要的部份是传播行为。）有以下选项可供使用：

* PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
* PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。
* PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。
* PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。
* PROPAGATION\_NOT\_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
* PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

### SpringBoot

#### 什么是SpringBoot

在您第1次接触和学习Spring框架的时候，是否因为其繁杂的配置而退却了？在你第n次使用Spring框架的时候，是否觉得一堆反复黏贴的配置有一些厌烦？那么您就不妨来试试使用Spring Boot来让你更易上手，更简单快捷地构建Spring应用！

Spring Boot让我们的Spring应用变的更轻量化。比如：你可以仅仅依靠一个Java类来运行一个Spring引用。你也可以打包你的应用为jar并通过使用java -jar来运行你的Spring Web应用。

Spring Boot的主要优点：

为所有Spring开发者更快的入门

开箱即用，提供各种默认配置来简化项目配置

内嵌式容器简化Web项目

没有冗余代码生成和XML配置的要求

本章主要目标完成Spring Boot基础项目的构建，并且实现一个简单的Http请求处理，通过这个例子对Spring Boot有一个初步的了解，并体验其结构简单、开发快速的特性。

#### SpringBoot优点

为所有Spring开发者更快的入门

开箱即用，提供各种默认配置来简化项目配置

内嵌式容器简化Web项目

没有冗余代码生成和XML配置的要求

本章主要目标完成Spring Boot基础项目的构建，并且实现一个简单的Http请求处理，通过这个例子对Spring Boot有一个初步的了解，并体验其结构简单、开发快速的特性。

#### @RestController

**在上加上RestController 表示修饰该Controller所有的方法返回JSON格式,直接可以编写**

**Restful接口**

#### @EnableAutoConfiguration

注解:作用在于让 Spring Boot   根据应用所声明的依赖来对 Spring 框架进行自动配置  
        这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于spring-boot-starter-web添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用并相应地对Spring进行设置。

#### SpringApplication.run(HelloController.class, args);

标识为启动类

### Mybatis与Hibernate区别

**Mybatis是轻量级封装,Hibernate是重量级封装**

**Mybatis 以SQL语句得到对象,hibernate是以对象得到SQL语句**

### Mybatis#与$区别

**优先使用 #{}。因为 ${} 会导致 sql 注入的问题**

## 安全与防御部分

### 表单重复提交解决方案(防止Http重复提交。)

**产生原因**

**网络延时、重新刷新、点击浏览器的【后退】按钮回退到表单页面后进行再次提交**

**使用javascript 解决**

**既然存在上述所说的表单重复提交问题，那么我们就要想办法解决，比较常用的方法是采用JavaScript来防止表单重复提交，具体做法如下：**

**修改form.jsp页面，添加如下的JavaScript代码来防止表单重复提交**

代码:

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<title>Form表单</title>**  **<script type=*"text/javascript"*>**  **var isFlag = false;** **//表单是否已经提交标识，默认为false**  **function submitFlag() {**  **if (isFlag == false) {**  **isFlag = true;**  **return true;**  **} else {**  **return false;**  **}**  **}**  **</script>**  **</head>**  **<body>**  **<form action=*"*${pageContext.request.contextPath}*/DoFormServlet"***  **method=*"post"* onsubmit="return submitFlag()">**  **用户名：<input type=*"text"* name=*"userName"*> <input type=*"submit"***  **value=*"提交"* id=*"submit"*>**  **</form>**  **</body>**  **</html>** |

**除了用这种方式之外，经常见的另一种方式就是表单提交之后，将提交按钮设置为不可用，让用户没有机会点击第二次提交按钮，代码如下：**

|  |
| --- |
| function dosubmit(){  //获取表单提交按钮  var btnSubmit = document.getElementById("submit");  //将表单提交按钮设置为不可用，这样就可以避免用户再次点击提交按钮  btnSubmit.disabled= "disabled";  //返回true让表单可以正常提交  return true;  } |

#### 6使用后端提交解决

**对于【场景二】和【场景三】导致表单重复提交的问题，既然客户端无法解决，那么就在服务器端解决，在服务器端解决就需要用到session了。**

**具体的做法：在服务器端生成一个唯一的随机标识号，专业术语称为Token(令牌)，同时在当前用户的Session域中保存这个Token。然后将Token发送到客户端的Form表单中，在Form表单中使用隐藏域来存储这个Token，表单提交的时候连同这个Token一起提交到服务器端，然后在服务器端判断客户端提交上来的Token与服务器端生成的Token是否一致，如果不一致，那就是重复提交了，此时服务器端就可以不处理重复提交的表单。如果相同则处理表单提交，处理完后清除当前用户的Session域中存储的标识号。  
　　在下列情况下，服务器程序将拒绝处理用户提交的表单请求：**

**存储Session域中的Token(令牌)与表单提交的Token(令牌)不同。**

**当前用户的Session中不存在Token(令牌)。**

**用户提交的表单数据中没有Token(令牌)。**

**转发代码:**

|  |
| --- |
| **@WebServlet("/ForwardServlet")**  **public class ForwardServlet extends HttpServlet {**  **@Override**  **protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {**  **req.getSession().setAttribute("sesionToken", TokenUtils.*getToken*());**  **req.getRequestDispatcher("form.jsp").forward(req, resp);**  **}**  **}** |

转发页面:

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<title>Form表单</title>**  **</head>**  **<body>**  **<form action=*"*${pageContext.request.contextPath}*/DoFormServlet"***  **method=*"post"* onsubmit="return dosubmit()">**  **<input type=*"hidden"* name=*"token"* value=*"*${sesionToken}*"*> 用户名：<input type=*"text"***  **name=*"userName"*> <input type=*"submit"* value=*"提交"* id=*"submit"*>**  **</form>**  **</body>**  **</html>** |

后端Java代码:

|  |
| --- |
| @WebServlet("/DoFormServlet")  **public** **class** DoFormServlet **extends** HttpServlet {  @Override  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.setCharacterEncoding("UTF-8");  **boolean** flag = isFlag(req);  **if** (!flag) {  resp.getWriter().write("已经提交...");  System.***out***.println("数据已经提交了..");  **return**;  }  String userName = req.getParameter("userName");  **try** {  Thread.*sleep*(300);  } **catch** (Exception e) {  // **TODO**: handle exception  }  System.***out***.println("往数据库插入数据...." + userName);  resp.getWriter().write("success");  }  **public** **boolean** isFlag(HttpServletRequest request) {  HttpSession session = request.getSession();  String sesionToken = (String) session.getAttribute("sesionToken");  String token = request.getParameter("token");  **if** (!(token.equals(sesionToken))) {  **return** **false**;  }  session.removeAttribute("sesionToken");  **return** **true**;  }  } |

### 如何防御XSS攻击

**XSS攻击使用Javascript脚本注入进行攻击**

**解决办法:就是将请求可能会发送的特殊字符、javascript标签转换为html代码执行**

**例如在表单中注入: <script>location.href='http://www.itmayiedu.com'</script>**

**注意:谷歌浏览器 已经防止了XSS攻击，为了演示效果，最好使用火狐浏览器**

**实例:**

**演示:**

**代码: fromToXss.jsp**

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>**  **<title>Insert title here</title>**  **</head>**  **<body>**  **<form action=*"XssDemo"* method=*"post"*>**  **<input type=*"text"* name=*"userName"*> <input type=*"submit"*>**  **</form>**  **</body>**  **</html>** |

代码: XssDemo

|  |
| --- |
| **import** java.io.IOException;  **import** javax.servlet.ServletException;  **import** javax.servlet.annotation.WebServlet;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  @WebServlet("/XssDemo")  **public** **class** XssDemo **extends** HttpServlet {  @Override  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  String userName = req.getParameter("userName");  req.setAttribute("userName", userName);  req.getRequestDispatcher("showUserName.jsp").forward(req, resp);  }    } |

代码: showUserName.jsp

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>**  **<title>Insert title here</title>**  **</head>**  **<body>userName:${userName}**  **</body>**  **</html>** |

**7.1 解決方案**

**使用Fileter过滤器过滤器注入标签**

**FilterDemo**

|  |
| --- |
| **import java.io.IOException;**  **import java.util.Map;**  **import javax.servlet.Filter;**  **import javax.servlet.FilterChain;**  **import javax.servlet.FilterConfig;**  **import javax.servlet.ServletException;**  **import javax.servlet.ServletRequest;**  **import javax.servlet.ServletResponse;**  **import javax.servlet.annotation.WebFilter;**  **import javax.servlet.http.HttpServletRequest;**  **import javax.servlet.http.HttpServletResponse;**  **/\*\***  **\* 使用Filter 打印参数**  **\***  **\* @author Administrator**  **\***  **\*/**  **public class FilterDemo implements Filter {**  **public FilterDemo() {**  **System.*out*.println("FilterDemo 构造函数被执行...");**  **}**  **/\*\***  **\* 销毁**  **\*/**  **public void destroy() {**  **System.*out*.println("destroy");**  **}**  **public void doFilter(ServletRequest paramServletRequest, ServletResponse paramServletResponse,**  **FilterChain paramFilterChain) throws IOException, ServletException {**  **System.*out*.println("doFilter");**  **HttpServletRequest request = (HttpServletRequest) paramServletRequest;**  **XssAndSqlHttpServletRequestWrapper xssRequestWrapper = new XssAndSqlHttpServletRequestWrapper(request);**  **paramFilterChain.doFilter(xssRequestWrapper, paramServletResponse);**  **}**  **/\*\***  **\* 初始化**  **\*/**  **public void init(FilterConfig paramFilterConfig) throws ServletException {**  **System.*out*.println("init");**  **}**  **}** |

**XssAndSqlHttpServletRequestWrapper**

|  |
| --- |
| **import javax.servlet.http.HttpServletRequest;**  **import javax.servlet.http.HttpServletRequestWrapper;**  **import org.apache.commons.lang3.StringEscapeUtils;**  **import org.apache.commons.lang3.StringUtils;**  **/\*\***  **\* 防止XSS攻击**  **\*/**  **public class XssAndSqlHttpServletRequestWrapper extends HttpServletRequestWrapper {**  **HttpServletRequest request;**  **public XssAndSqlHttpServletRequestWrapper(HttpServletRequest request) {**  **super(request);**  **this.request = request;**  **}**  **@Override**  **public String getParameter(String name) {**  **String value = request.getParameter(name);**  **System.*out*.println("name:" + name + "," + value);**  **if (!StringUtils.*isEmpty*(value)) {**  **// 转换Html**  **value = StringEscapeUtils.*escapeHtml4*(value);**  **}**  **return value;**  **}**  **}** |

### 跨域实战解决方案

**跨域原因产生：在当前域名请求网站中，默认不允许通过ajax请求发送其他域名。**

**XMLHttpRequest cannot load 跨域问题解决办法**

#### 使用后台response添加header

**后台response添加header，response.setHeader("Access-Control-Allow-Origin", "\*"); 支持所有网站**

#### 使用JSONP

**JSONP的优缺点:**

**JSONP只支持get请求不支持psot请求**

**什么是SQL语句注入**

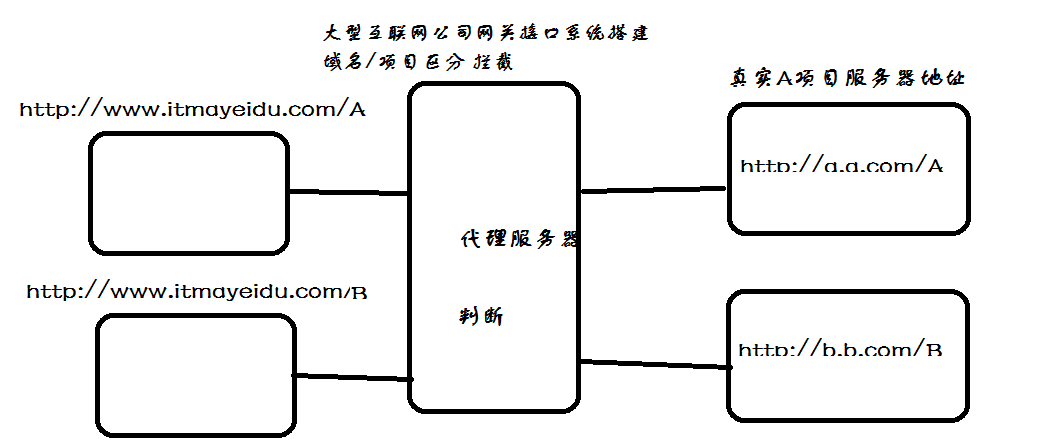
#### 使用接口网关

**使用nginx转发。**

**配置:**

|  |
| --- |
| server {  listen 80;  server\_name www.itmayiedu.com;  location /A {  proxy\_pass http://a.a.com:81/A;  index index.html index.htm;  }  location /B {  proxy\_pass http://b.b.com:81/B;  index index.html index.htm;  }  } |

**相关图:**



#### 使用内部服务器转发

**内部服务器使用HttpClient技术进行转发**

### 什么是SQL语句注入

**Sql语句如果是通过拼接方式执行的话,传入参数 ‘ or 1=1 会发生语句成立，导致数据错误。**

**应该使用PreparedStatement 先编译 在执行 通过？号穿参数的方式进行执行。**

### 怎么防御DDOC？

#### nginx配置DDOS

##### 限制请求速度

设置Nginx、Nginx Plus的连接请求在一个真实用户请求的合理范围内。比如，如果你觉得一个正常用户每两秒可以请求一次登录页面，你就可以设置Nginx每两秒钟接收一个客户端IP的请求（大约等同于每分钟30个请求）。

|  |
| --- |
| limit\_req\_zone $binary\_remote\_addr zone=one:10m rate=30r/m;  server {  ...  location /login.html {  limit\_req zone=one;  ...  }  } |

`limit\_req\_zone`命令设置了一个叫one的共享内存区来存储请求状态的特定键值，在上面的例子中是客户端IP($binary\_remote\_addr)。location块中的`limit\_req`通过引用one共享内存区来实现限制访问/login.html的目的。

##### 限制请求速度

设置Nginx、Nginx Plus的连接数在一个真实用户请求的合理范围内。比如，你可以设置每个客户端IP连接/store不可以超过10个。

## 缓存部分

### 什么是NOSQL?

**NoSQL 是 Not Only SQL 的缩写，意即"不仅仅是SQL"的意思，泛指非关系型的数据库。强调Key-Value Stores和文档数据库的优点，而不是单纯的反对RDBMS。**

**NoSQL产品是传统关系型数据库的功能阉割版本，通过减少用不到或很少用的功能，来大幅度提高产品性能**

**NoSQL产品 redis、mongodb** [**Membase**](http://www.couchbase.org/membase)**、**[**HBase**](http://hbase.apache.org/)

### 什么是Redis?

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：

Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。

Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

Redis支持数据的备份，即master-slave模式的数据备份。

### Redis应用场景

主要能够体现 解决数据库的访问压力。

例如:短信验证码时间有效期、session共享解决方案

### Redis优势

性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。

丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。

丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

### Redis与其他key-value存储有什么不同？

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

### Redis的基本数据类型

#### 字符串类型(String)

|  |
| --- |
| redis 127.0.0.1:6379> SET mykey "redis"  OK  redis 127.0.0.1:6379> GET mykey  "redis" |

在上面的例子中，SET和GET是redis中的命令，而mykey是键的名称。

Redis字符串命令用于管理Redis中的字符串值。以下是使用Redis字符串命令的语法。

redis 127.0.0.1:6379> COMMAND KEY\_NAME

Shell

**示例**

redis 127.0.0.1:6379> SET mykey "redis"

OK

redis 127.0.0.1:6379> GET mykey

"redis"

Shell

在上面的例子中，SET和GET是redis中的命令，而mykey是键的名称。

Redis字符串命令

下表列出了一些用于在**Redis**中管理字符串的基本命令。

| **编号** | **命令** | **描述说明** |
| --- | --- | --- |
| 1 | [SET key value](http://www.yiibai.com/redis/strings_set.html) | 此命令设置指定键的值。 |
| 2 | [GET key](http://www.yiibai.com/redis/strings_get.html) | 获取指定键的值。 |
| 3 | [GETRANGE key start end](http://www.yiibai.com/redis/strings_getrange.html) | 获取存储在键上的字符串的子字符串。 |
| 4 | [GETSET key value](http://www.yiibai.com/redis/strings_getset.html) | 设置键的字符串值并返回其旧值。 |
| 5 | [GETBIT key offset](http://www.yiibai.com/redis/strings_getbit.html) | 返回在键处存储的字符串值中偏移处的位值。 |
| 6 | [MGET key1 [key2..]](http://www.yiibai.com/redis/strings_mget.html) | 获取所有给定键的值 |
| 7 | [SETBIT key offset value](http://www.yiibai.com/redis/strings_setbit.html) | 存储在键上的字符串值中设置或清除偏移处的位 |
| 8 | [SETEX key seconds value](http://www.yiibai.com/redis/strings_setex.html) | 使用键和到期时间来设置值 |
| 9 | [SETNX key value](http://www.yiibai.com/redis/strings_setnx.html) | 设置键的值，仅当键不存在时 |
| 10 | [SETRANGE key offset value](http://www.yiibai.com/redis/strings_setrange.html) | 在指定偏移处开始的键处覆盖字符串的一部分 |
| 11 | [STRLEN key](http://www.yiibai.com/redis/strings_strlen.html) | 获取存储在键中的值的长度 |
| 12 | [MSET key value [key value …]](http://www.yiibai.com/redis/strings_mset.html) | 为多个键分别设置它们的值 |
| 13 | [MSETNX key value [key value …]](http://www.yiibai.com/redis/strings_msetnx.html) | 为多个键分别设置它们的值，仅当键不存在时 |
| 14 | [PSETEX key milliseconds value](http://www.yiibai.com/redis/strings_psetex.html) | 设置键的值和到期时间(以毫秒为单位) |
| 15 | [INCR key](http://www.yiibai.com/redis/strings_incr.html) | 将键的整数值增加1 |
| 16 | [INCRBY key increment](http://www.yiibai.com/redis/strings_incrby.html) | 将键的整数值按给定的数值增加 |
| 17 | [INCRBYFLOAT key increment](http://www.yiibai.com/redis/strings_incrbyfloat.html) | 将键的浮点值按给定的数值增加 |
| 18 | [DECR key](http://www.yiibai.com/redis/strings_decr.html) | 将键的整数值减1 |
| 19 | [DECRBY key decrement](http://www.yiibai.com/redis/strings_decrby.html) | 按给定数值减少键的整数值 |
| 20 | [APPEND key value](http://www.yiibai.com/redis/strings_append.html) | 将指定值附加到键 |

#### 列表类型(List)

Redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）

一个列表最多可以包含 232 - 1 个元素 (4294967295, 每个列表超过40亿个元素)。

redis 127.0.0.1:6379> LPUSH runoobkey redis

(integer) 1

redis 127.0.0.1:6379> LPUSH runoobkey mongodb

(integer) 2

redis 127.0.0.1:6379> LPUSH runoobkey mysql

(integer) 3

redis 127.0.0.1:6379> LRANGE runoobkey 0 10

1) "mysql"

2) "mongodb"

3) "redis"

Redis 列表命令

下表列出了列表相关的基本命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [BLPOP key1 [key2 ] timeout](http://www.runoob.com/redis/lists-blpop.html)  移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。 |
| 2 | [BRPOP key1 [key2 ] timeout](http://www.runoob.com/redis/lists-brpop.html)  移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。 |
| 3 | [BRPOPLPUSH source destination timeout](http://www.runoob.com/redis/lists-brpoplpush.html)  从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。 |
| 4 | [LINDEX key index](http://www.runoob.com/redis/lists-lindex.html)  通过索引获取列表中的元素 |
| 5 | [LINSERT key BEFORE|AFTER pivot value](http://www.runoob.com/redis/lists-linsert.html)  在列表的元素前或者后插入元素 |
| 6 | [LLEN key](http://www.runoob.com/redis/lists-llen.html)  获取列表长度 |
| 7 | [LPOP key](http://www.runoob.com/redis/lists-lpop.html)  移出并获取列表的第一个元素 |
| 8 | [LPUSH key value1 [value2]](http://www.runoob.com/redis/lists-lpush.html)  将一个或多个值插入到列表头部 |
| 9 | [LPUSHX key value](http://www.runoob.com/redis/lists-lpushx.html)  将一个值插入到已存在的列表头部 |
| 10 | [LRANGE key start stop](http://www.runoob.com/redis/lists-lrange.html)  获取列表指定范围内的元素 |
| 11 | [LREM key count value](http://www.runoob.com/redis/lists-lrem.html)  移除列表元素 |
| 12 | [LSET key index value](http://www.runoob.com/redis/lists-lset.html)  通过索引设置列表元素的值 |
| 13 | [LTRIM key start stop](http://www.runoob.com/redis/lists-ltrim.html)  对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。 |
| 14 | [RPOP key](http://www.runoob.com/redis/lists-rpop.html)  移除并获取列表最后一个元素 |
| 15 | [RPOPLPUSH source destination](http://www.runoob.com/redis/lists-rpoplpush.html)  移除列表的最后一个元素，并将该元素添加到另一个列表并返回 |
| 16 | [RPUSH key value1 [value2]](http://www.runoob.com/redis/lists-rpush.html)  在列表中添加一个或多个值 |
| 17 | [RPUSHX key value](http://www.runoob.com/redis/lists-rpushx.html)  为已存在的列表添加值 |

#### 集合(Set)

Redis的Set是string类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

Redis 中 集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

**实例**

redis 127.0.0.1:6379> SADD runoobkey redis

(integer) 1

redis 127.0.0.1:6379> SADD runoobkey mongodb

(integer) 1

redis 127.0.0.1:6379> SADD runoobkey mysql

(integer) 1

redis 127.0.0.1:6379> SADD runoobkey mysql

(integer) 0

redis 127.0.0.1:6379> SMEMBERS runoobkey

1) "mysql"

2) "mongodb"

3) "redis"

在以上实例中我们通过 **SADD** 命令向名为 **runoobkey** 的集合插入的三个元素。

**Redis 集合命令**

下表列出了 Redis 集合基本命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [SADD key member1 [member2]](http://www.runoob.com/redis/sets-sadd.html)  向集合添加一个或多个成员 |
| 2 | [SCARD key](http://www.runoob.com/redis/sets-scard.html)  获取集合的成员数 |
| 3 | [SDIFF key1 [key2]](http://www.runoob.com/redis/sets-sdiff.html)  返回给定所有集合的差集 |
| 4 | [SDIFFSTORE destination key1 [key2]](http://www.runoob.com/redis/sets-sdiffstore.html)  返回给定所有集合的差集并存储在 destination 中 |
| 5 | [SINTER key1 [key2]](http://www.runoob.com/redis/sets-sinter.html)  返回给定所有集合的交集 |
| 6 | [SINTERSTORE destination key1 [key2]](http://www.runoob.com/redis/sets-sinterstore.html)  返回给定所有集合的交集并存储在 destination 中 |
| 7 | [SISMEMBER key member](http://www.runoob.com/redis/sets-sismember.html)  判断 member 元素是否是集合 key 的成员 |
| 8 | [SMEMBERS key](http://www.runoob.com/redis/sets-smembers.html)  返回集合中的所有成员 |
| 9 | [SMOVE source destination member](http://www.runoob.com/redis/sets-smove.html)  将 member 元素从 source 集合移动到 destination 集合 |
| 10 | [SPOP key](http://www.runoob.com/redis/sets-spop.html)  移除并返回集合中的一个随机元素 |
| 11 | [SRANDMEMBER key [count]](http://www.runoob.com/redis/sets-srandmember.html)  返回集合中一个或多个随机数 |
| 12 | [SREM key member1 [member2]](http://www.runoob.com/redis/sets-srem.html)  移除集合中一个或多个成员 |
| 13 | [SUNION key1 [key2]](http://www.runoob.com/redis/sets-sunion.html)  返回所有给定集合的并集 |
| 14 | [SUNIONSTORE destination key1 [key2]](http://www.runoob.com/redis/sets-sunionstore.html)  所有给定集合的并集存储在 destination 集合中 |
| 15 | [SSCAN key cursor [MATCH pattern] [COUNT count]](http://www.runoob.com/redis/sets-sscan.html)  迭代集合中的元素 |

#### 有序集合(sorted set)

Redis 有序集合和集合一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

有序集合的成员是唯一的,但分数(score)却可以重复。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。 集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

**实例**

redis 127.0.0.1:6379> ZADD runoobkey 1 redis

(integer) 1

redis 127.0.0.1:6379> ZADD runoobkey 2 mongodb

(integer) 1

redis 127.0.0.1:6379> ZADD runoobkey 3 mysql

(integer) 1

redis 127.0.0.1:6379> ZADD runoobkey 3 mysql

(integer) 0

redis 127.0.0.1:6379> ZADD runoobkey 4 mysql

(integer) 0

redis 127.0.0.1:6379> ZRANGE runoobkey 0 10 WITHSCORES

1) "redis"

2) "1"

3) "mongodb"

4) "2"

5) "mysql"

6) "4"

在以上实例中我们通过命令 **ZADD** 向 redis 的有序集合中添加了三个值并关联上分数。

**Redis 有序集合命令**

下表列出了 redis 有序集合的基本命令:

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [ZADD key score1 member1 [score2 member2]](http://www.runoob.com/redis/sorted-sets-zadd.html)  向有序集合添加一个或多个成员，或者更新已存在成员的分数 |
| 2 | [ZCARD key](http://www.runoob.com/redis/sorted-sets-zcard.html)  获取有序集合的成员数 |
| 3 | [ZCOUNT key min max](http://www.runoob.com/redis/sorted-sets-zcount.html)  计算在有序集合中指定区间分数的成员数 |
| 4 | [ZINCRBY key increment member](http://www.runoob.com/redis/sorted-sets-zincrby.html)  有序集合中对指定成员的分数加上增量 increment |
| 5 | [ZINTERSTORE destination numkeys key [key ...]](http://www.runoob.com/redis/sorted-sets-zinterstore.html)  计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中 |
| 6 | [ZLEXCOUNT key min max](http://www.runoob.com/redis/sorted-sets-zlexcount.html)  在有序集合中计算指定字典区间内成员数量 |
| 7 | [ZRANGE key start stop [WITHSCORES]](http://www.runoob.com/redis/sorted-sets-zrange.html)  通过索引区间返回有序集合成指定区间内的成员 |
| 8 | [ZRANGEBYLEX key min max [LIMIT offset count]](http://www.runoob.com/redis/sorted-sets-zrangebylex.html)  通过字典区间返回有序集合的成员 |
| 9 | [ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]](http://www.runoob.com/redis/sorted-sets-zrangebyscore.html)  通过分数返回有序集合指定区间内的成员 |
| 10 | [ZRANK key member](http://www.runoob.com/redis/sorted-sets-zrank.html)  返回有序集合中指定成员的索引 |
| 11 | [ZREM key member [member ...]](http://www.runoob.com/redis/sorted-sets-zrem.html)  移除有序集合中的一个或多个成员 |
| 12 | [ZREMRANGEBYLEX key min max](http://www.runoob.com/redis/sorted-sets-zremrangebylex.html)  移除有序集合中给定的字典区间的所有成员 |
| 13 | [ZREMRANGEBYRANK key start stop](http://www.runoob.com/redis/sorted-sets-zremrangebyrank.html)  移除有序集合中给定的排名区间的所有成员 |
| 14 | [ZREMRANGEBYSCORE key min max](http://www.runoob.com/redis/sorted-sets-zremrangebyscore.html)  移除有序集合中给定的分数区间的所有成员 |
| 15 | [ZREVRANGE key start stop [WITHSCORES]](http://www.runoob.com/redis/sorted-sets-zrevrange.html)  返回有序集中指定区间内的成员，通过索引，分数从高到底 |
| 16 | [ZREVRANGEBYSCORE key max min [WITHSCORES]](http://www.runoob.com/redis/sorted-sets-zrevrangebyscore.html)  返回有序集中指定分数区间内的成员，分数从高到低排序 |
| 17 | [ZREVRANK key member](http://www.runoob.com/redis/sorted-sets-zrevrank.html)  返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序 |
| 18 | [ZSCORE key member](http://www.runoob.com/redis/sorted-sets-zscore.html)  返回有序集中，成员的分数值 |
| 19 | [ZUNIONSTORE destination numkeys key [key ...]](http://www.runoob.com/redis/sorted-sets-zunionstore.html)  计算给定的一个或多个有序集的并集，并存储在新的 key 中 |
| 20 | [ZSCAN key cursor [MATCH pattern] [COUNT count]](http://www.runoob.com/redis/sorted-sets-zscan.html)  迭代有序集合中的元素（包括元素成员和元素分值） |

#### 哈希(Hash)

Redis hash 是一个string类型的field和value的映射表，hash特别适合用于存储对象。

Redis 中每个 hash 可以存储 232 - 1 键值对（40多亿）。

**实例**

127.0.0.1:6379> HMSET runoobkey name "redis tutorial" description "redis basic commands for caching" likes 20 visitors 23000

OK

127.0.0.1:6379> HGETALL runoobkey

1) "name"

2) "redis tutorial"

3) "description"

4) "redis basic commands for caching"

5) "likes"

6) "20"

7) "visitors"

8) "23000"

**hset key mapHey MapValue**

在以上实例中，我们设置了 redis 的一些描述信息(name, description, likes, visitors) 到哈希表的 **runoobkey** 中。

**Redis hash 命令**

下表列出了 redis hash 基本的相关命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [HDEL key field2 [field2]](http://www.runoob.com/redis/hashes-hdel.html)  删除一个或多个哈希表字段 |
| 2 | [HEXISTS key field](http://www.runoob.com/redis/hashes-hexists.html)  查看哈希表 key 中，指定的字段是否存在。 |
| 3 | [HGET key field](http://www.runoob.com/redis/hashes-hget.html)  获取存储在哈希表中指定字段的值。 |
| 4 | [HGETALL key](http://www.runoob.com/redis/hashes-hgetall.html)  获取在哈希表中指定 key 的所有字段和值 |
| 5 | [HINCRBY key field increment](http://www.runoob.com/redis/hashes-hincrby.html)  为哈希表 key 中的指定字段的整数值加上增量 increment 。 |
| 6 | [HINCRBYFLOAT key field increment](http://www.runoob.com/redis/hashes-hincrbyfloat.html)  为哈希表 key 中的指定字段的浮点数值加上增量 increment 。 |
| 7 | [HKEYS key](http://www.runoob.com/redis/hashes-hkeys.html)  获取所有哈希表中的字段 |
| 8 | [HLEN key](http://www.runoob.com/redis/hashes-hlen.html)  获取哈希表中字段的数量 |
| 9 | [HMGET key field1 [field2]](http://www.runoob.com/redis/hashes-hmget.html)  获取所有给定字段的值 |
| 10 | [HMSET key field1 value1 [field2 value2 ]](http://www.runoob.com/redis/hashes-hmset.html)  同时将多个 field-value (域-值)对设置到哈希表 key 中。 |
| 11 | [HSET key field value](http://www.runoob.com/redis/hashes-hset.html)  将哈希表 key 中的字段 field 的值设为 value 。 |
| 12 | [HSETNX key field value](http://www.runoob.com/redis/hashes-hsetnx.html)  只有在字段 field 不存在时，设置哈希表字段的值。 |
| 13 | [HVALS key](http://www.runoob.com/redis/hashes-hvals.html)  获取哈希表中所有值 |
| 14 | HSCAN key cursor [MATCH pattern] [COUNT count]  迭代哈希表中的键值对。 |

### 什么是redis的主从复制

概述

1、redis的复制功能是支持多个数据库之间的数据同步。一类是主数据库（master）一类是从数据库（slave），主数据库可以进行读写操作，当发生写操作的时候自动将数据同步到从数据库，而从数据库一般是只读的，并接收主数据库同步过来的数据，一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

2、通过redis的复制功能可以很好的实现数据库的读写分离，提高服务器的负载能力。主数据库主要进行写操作，而从数据库负责读操作。

主从复制过程

主从复制过程：见下图



过程：

1：当一个从数据库启动时，会向主数据库发送sync命令，

2：主数据库接收到sync命令后会开始在后台保存快照（执行rdb操作），并将保存期间接收到的命令缓存起来

3：当快照完成后，redis会将快照文件和所有缓存的命令发送给从数据库。

4：从数据库收到后，会载入快照文件并执行收到的缓存的命令。

修改redis.conf

修改从redis中的 redis.conf文件

slaveof 192.168.33.130 6379

masterauth 123456--- 主redis服务器配置了密码,则需要配置

### 什么是哨兵机制

Redis的哨兵(sentinel) 系统用于管理多个 Redis 服务器,该系统执行以下三个任务:

·        **监控(Monitoring)**: 哨兵(sentinel) 会不断地检查你的Master和Slave是否运作正常。

·        **提醒(Notification)**:当被监控的某个 Redis出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。

·        **自动故障迁移(Automatic failover)**:当一个Master不能正常工作时，哨兵(sentinel) 会开始一次自动故障迁移操作,它会将失效Master的其中一个Slave升级为新的Master, 并让失效Master的其他Slave改为复制新的Master; 当客户端试图连接失效的Master时,集群也会向客户端返回新Master的地址,使得集群可以使用Master代替失效Master。

哨兵(sentinel) 是一个分布式系统,你可以在一个架构中运行多个哨兵(sentinel) 进程,这些进程使用流言协议(gossipprotocols)来接收关于Master是否下线的信息,并使用投票协议(agreement protocols)来决定是否执行自动故障迁移,以及选择哪个Slave作为新的Master.

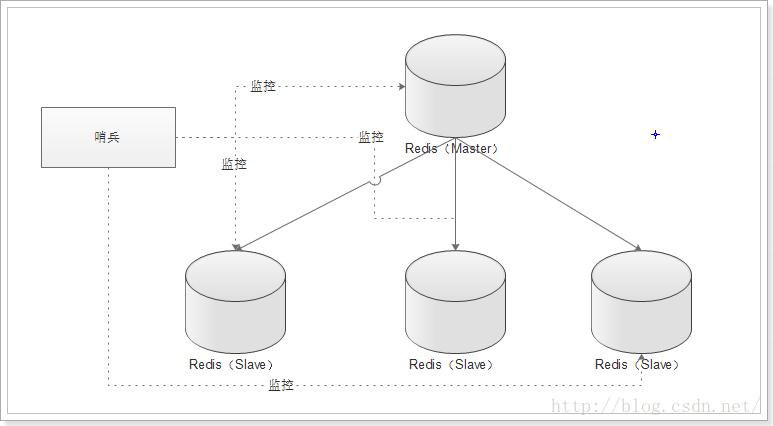
每个哨兵(sentinel) 会向其它哨兵(sentinel)、master、slave定时发送消息,以确认对方是否”活”着,如果发现对方在指定时间(可配置)内未回应,则暂时认为对方已挂(**所谓的”主观认为宕机” Subjective Down,简称sdown**).

若“哨兵群”中的多数sentinel,都报告某一master没响应,系统才认为该master"彻底死亡"(**即:客观上的真正down机,Objective Down,简称odown**),通过一定的vote算法,从剩下的slave节点中,选一台提升为master,然后自动修改相关配置.

虽然哨兵(sentinel) 释出为一个单独的可执行文件 redis-sentinel ,但实际上它只是一个运行在特殊模式下的 Redis 服务器，你可以在启动一个普通 Redis 服务器时通过给定 --sentinel 选项来启动哨兵(sentinel).

哨兵(sentinel) 的一些设计思路和zookeeper非常类似

单个哨兵(sentinel)



10.2 哨兵模式修改配置

实现步骤:

1.拷贝到etc目录

cp sentinel.conf /usr/local/redis/etc

2.修改sentinel.conf配置文件

sentinel monitor mymast 192.168.110.133 6379 1 #主节点 名称 IP 端口号 选举次数

3. 修改心跳检测 5000毫秒

sentinel down-after-milliseconds mymaster 5000

4.sentinel parallel-syncs mymaster 2 --- 做多多少合格节点

5. 启动哨兵模式

./redis-server /usr/local/redis/etc/sentinel.conf --sentinel &

6. 停止哨兵模式

### Redis事物

Redis 事务可以一次执行多个命令， 并且带有以下两个重要的保证：

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

一个事务从开始到执行会经历以下三个阶段：

开始事务。

命令入队。

执行事务。

以下是一个事务的例子， 它先以 MULTI 开始一个事务， 然后将多个命令入队到事务中， 最后由 EXEC 命令触发事务， 一并执行事务中的所有命令：

|  |
| --- |
| redis 127.0.0.1:6379> MULTI  OK  redis 127.0.0.1:6379> SET book-name "Mastering C++ in 21 days"  QUEUED  redis 127.0.0.1:6379> GET book-name  QUEUED  redis 127.0.0.1:6379> SADD tag "C++" "Programming" "Mastering Series"  QUEUED  redis 127.0.0.1:6379> SMEMBERS tag  QUEUED  redis 127.0.0.1:6379> EXEC  1) OK  2) "Mastering C++ in 21 days"  3) (integer) 3  4) 1) "Mastering Series"  2) "C++"  3) "Programming" |

### Redis持久化

什么是Redis持久化,就是将内存数据保存到硬盘。

Redis 持久化存储 (AOF 与 RDB 两种模式)

#### RDB持久化

RDB 是在某个时间点将数据写入一个临时文件，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复。  
优点：**使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能**  
缺点：**RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候**

这里说的这个执行数据写入到临时文件的时间点是可以通过配置来自己确定的，通过配置**redis 在 n 秒内如果超过 m 个 key 被修改**这执行一次 RDB 操作。这个操作就类似于在这个时间点来保存一次 Redis 的所有数据，一次快照数据。所有这个持久化方法也通常叫做 snapshots。

RDB 默认开启，redis.conf 中的具体配置参数如下；

|  |
| --- |
| *#dbfilename：持久化数据存储在本地的文件*  dbfilename dump.rdb  *#dir：持久化数据存储在本地的路径，如果是在/redis/redis-3.0.6/src下启动的redis-cli，则数据会存储在当前src目录下*  dir ./  *##snapshot触发的时机，save*  *##如下为900秒后，至少有一个变更操作，才会snapshot*  *##对于此值的设置，需要谨慎，评估系统的变更操作密集程度*  *##可以通过“save “””来关闭snapshot功能*  *#save时间，以下分别表示更改了1个key时间隔900s进行持久化存储；更改了10个key300s进行存储；更改10000个key60s进行存储。*  save 900 1  save 300 10  save 60 10000  *##当snapshot时出现错误无法继续时，是否阻塞客户端“变更操作”，“错误”可能因为磁盘已满/磁盘故障/OS级别异常等*  stop-writes-on-bgsave-error yes  *##是否启用rdb文件压缩，默认为“yes”，压缩往往意味着“额外的cpu消耗”，同时也意味这较小的文件尺寸以及较短的网络传输时间*  rdbcompression yes |

#### AOF持久化

Append-only file，将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在 append 操作返回后(已经写入到文件或者即将写入)，才进行实际的数据变更，“日志文件”保存了历史所有的操作过程；当 server 需要数据恢复时，可以直接 replay 此日志文件，即可还原所有的操作过程。AOF 相对可靠，它和 mysql 中 bin.log、apache.log、zookeeper 中 txn-log 简直异曲同工。AOF 文件内容是字符串，非常容易阅读和解析。  
优点：可以保持更高的数据完整性，如果设置追加 file 的时间是 1s，如果 redis 发生故障，最多会丢失 1s 的数据；且如果日志写入不完整支持 redis-check-aof 来进行日志修复；AOF 文件没被 rewrite 之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的 flushall）。  
缺点：AOF 文件比 RDB 文件大，且恢复速度慢。

我们可以简单的认为 AOF 就是日志文件，此文件只会记录“变更操作”(例如：set/del 等)，如果 server 中持续的大量变更操作，将会导致 AOF 文件非常的庞大，意味着 server 失效后，数据恢复的过程将会很长；事实上，一条数据经过多次变更，将会产生多条 AOF 记录，其实只要保存当前的状态，历史的操作记录是可以抛弃的；因为 AOF 持久化模式还伴生了“AOF rewrite”。  
AOF 的特性决定了它相对比较安全，如果你期望数据更少的丢失，那么可以采用 AOF 模式。如果 AOF 文件正在被写入时突然 server 失效，有可能导致文件的最后一次记录是不完整，你可以通过手工或者程序的方式去检测并修正不完整的记录，以便通过 aof 文件恢复能够正常；同时需要提醒，如果你的 redis 持久化手段中有 aof，那么在 server 故障失效后再次启动前，需要检测 aof 文件的完整性。

AOF 默认关闭，开启方法，修改配置文件 reds.conf：appendonly yes

|  |
| --- |
| *##此选项为aof功能的开关，默认为“no”，可以通过“yes”来开启aof功能*  *##只有在“yes”下，aof重写/文件同步等特性才会生效*  appendonly yes  *##指定aof文件名称*  appendfilename appendonly.aof  *##指定aof操作中文件同步策略，有三个合法值：always everysec no,默认为everysec*  appendfsync everysec  *##在aof-rewrite期间，appendfsync是否暂缓文件同步，"no"表示“不暂缓”，“yes”表示“暂缓”，默认为“no”*  no-appendfsync-on-rewrite no  *##aof文件rewrite触发的最小文件尺寸(mb,gb),只有大于此aof文件大于此尺寸是才会触发rewrite，默认“64mb”，建议“512mb”*  auto-aof-rewrite-min-size 64mb  *##相对于“上一次”rewrite，本次rewrite触发时aof文件应该增长的百分比。*  *##每一次rewrite之后，redis都会记录下此时“新aof”文件的大小(例如A)，那么当aof文件增长到A\*(1 + p)之后*  *##触发下一次rewrite，每一次aof记录的添加，都会检测当前aof文件的尺寸。*  auto-aof-rewrite-percentage 100 |

AOF 是文件操作，对于变更操作比较密集的 server，那么必将造成磁盘 IO 的负荷加重；此外 linux 对文件操作采取了“延迟写入”手段，即并非每次 write 操作都会触发实际磁盘操作，而是进入了 buffer 中，当 buffer 数据达到阀值时触发实际写入(也有其他时机)，这是 linux 对文件系统的优化，但是这却有可能带来隐患，如果 buffer 没有刷新到磁盘，此时物理机器失效(比如断电)，那么有可能导致最后一条或者多条 aof 记录的丢失。通过上述配置文件，可以得知 redis 提供了 3 中 aof 记录同步选项：

always：每一条 aof 记录都立即同步到文件，这是最安全的方式，也以为更多的磁盘操作和阻塞延迟，是 IO 开支较大。

everysec：每秒同步一次，性能和安全都比较中庸的方式，也是 redis 推荐的方式。如果遇到物理服务器故障，有可能导致最近一秒内 aof 记录丢失(可能为部分丢失)。

no：redis 并不直接调用文件同步，而是交给操作系统来处理，操作系统可以根据 buffer 填充情况 / 通道空闲时间等择机触发同步；这是一种普通的文件操作方式。性能较好，在物理服务器故障时，数据丢失量会因 OS 配置有关。

其实，我们可以选择的太少，everysec 是最佳的选择。如果你非常在意每个数据都极其可靠，建议你选择一款“关系性数据库”吧。  
AOF 文件会不断增大，它的大小直接影响“故障恢复”的时间, 而且 AOF 文件中历史操作是可以丢弃的。AOF rewrite 操作就是“压缩”AOF 文件的过程，当然 redis 并没有采用“基于原 aof 文件”来重写的方式，而是采取了类似 snapshot 的方式：基于 copy-on-write，全量遍历内存中数据，然后逐个序列到 aof 文件中。因此 AOF rewrite 能够正确反应当前内存数据的状态，这正是我们所需要的；\*rewrite 过程中，对于新的变更操作将仍然被写入到原 AOF 文件中，同时这些新的变更操作也会被 redis 收集起来(buffer，copy-on-write 方式下，最极端的可能是所有的 key 都在此期间被修改，将会耗费 2 倍内存)，当内存数据被全部写入到新的 aof 文件之后，收集的新的变更操作也将会一并追加到新的 aof 文件中，此后将会重命名新的 aof 文件为 appendonly.aof, 此后所有的操作都将被写入新的 aof 文件。如果在 rewrite 过程中，出现故障，将不会影响原 AOF 文件的正常工作，只有当 rewrite 完成之后才会切换文件，因为 rewrite 过程是比较可靠的。\*

触发 rewrite 的时机可以通过配置文件来声明，同时 redis 中可以通过 bgrewriteaof 指令人工干预。

redis-cli -h ip -p port bgrewriteaof

因为 rewrite 操作 /aof 记录同步 /snapshot 都消耗磁盘 IO，redis 采取了“schedule”策略：无论是“人工干预”还是系统触发，snapshot 和 rewrite 需要逐个被执行。

AOF rewrite 过程并不阻塞客户端请求。系统会开启一个子进程来完成。

#### AOF与RDB区别

*OF 和 RDB 各有优缺点，这是有它们各自的特点所决定：*

1) AOF 更加安全，可以将数据更加及时的同步到文件中，但是 AOF 需要较多的磁盘 IO 开支，AOF 文件尺寸较大，文件内容恢复数度相对较慢。  
\*2) snapshot，安全性较差，它是“正常时期”数据备份以及 master-slave 数据同步的最佳手段，文件尺寸较小，恢复数度较快。

可以通过配置文件来指定它们中的一种，或者同时使用它们(不建议同时使用)，或者全部禁用，在架构良好的环境中，master 通常使用 AOF，slave 使用 snapshot，主要原因是 master 需要首先确保数据完整性，它作为数据备份的第一选择；slave 提供只读服务(目前 slave 只能提供读取服务)，它的主要目的就是快速响应客户端 read 请求；但是如果你的 redis 运行在网络稳定性差 / 物理环境糟糕情况下，建议你 master 和 slave 均采取 AOF，这个在 master 和 slave 角色切换时，可以减少“人工数据备份”/“人工引导数据恢复”的时间成本；如果你的环境一切非常良好，且服务需要接收密集性的 write 操作，那么建议 master 采取 snapshot，而 slave 采用 AOF。

### Redis发布订阅

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

下图展示了频道 channel1 ， 以及订阅这个频道的三个客户端 —— client2 、 client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时， 这个消息就会被发送给订阅它的三个客户端：



以下实例演示了发布订阅是如何工作的。在我们实例中我们创建了订阅频道名为 **redisChat**:

redis 127.0.0.1:6379> SUBSCRIBE redisChat

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "redisChat"

3) (integer) 1

现在，我们先重新开启个 redis 客户端，然后在同一个频道 redisChat 发布两次消息，订阅者就能接收到消息。

redis 127.0.0.1:6379> PUBLISH redisChat "Redis is a great caching technique"

(integer) 1

redis 127.0.0.1:6379> PUBLISH redisChat "Learn redis by runoob.com"

(integer) 1

# 订阅者的客户端会显示如下消息

1) "message"

2) "redisChat"

3) "Redis is a great caching technique"

1) "message"

2) "redisChat"

3) "Learn redis by runoob.com"

Redis 发布订阅命令

下表列出了 redis 发布订阅常用命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [PSUBSCRIBE pattern [pattern ...]](http://www.runoob.com/redis/pub-sub-psubscribe.html)  订阅一个或多个符合给定模式的频道。 |
| 2 | [PUBSUB subcommand [argument [argument ...]]](http://www.runoob.com/redis/pub-sub-pubsub.html)  查看订阅与发布系统状态。 |
| 3 | [PUBLISH channel message](http://www.runoob.com/redis/pub-sub-publish.html)  将信息发送到指定的频道。 |
| 4 | [PUNSUBSCRIBE [pattern [pattern ...]]](http://www.runoob.com/redis/pub-sub-punsubscribe.html)  退订所有给定模式的频道。 |
| 5 | [SUBSCRIBE channel [channel ...]](http://www.runoob.com/redis/pub-sub-subscribe.html)  订阅给定的一个或多个频道的信息。 |
| 6 | [UNSUBSCRIBE [channel [channel ...]]](http://www.runoob.com/redis/pub-sub-unsubscribe.html)  指退订给定的频道。 |

### Redis如何做集群？

## nginx

### 什么是nginx

nginx是一款高性能的http 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器。由俄罗斯的程序设计师Igor Sysoev所开发，官方测试nginx能够支支撑5万并发链接，并且cpu、内存等资源消耗却非常低，运行非常稳定，所以现在很多知名的公司都在使用nginx。

### nginx应用场景

1、http服务器。Nginx是一个http服务可以独立提供http服务。可以做网页静态服务器。

2、虚拟主机。可以实现在一台服务器虚拟出多个网站。例如个人网站使用的虚拟主机。

3、反向代理，负载均衡。当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使用nginx做反向代理。并且多台服务器可以平均分担负载，不会因为某台服务器负载高宕机而某台服务器闲置的情况。



### nginx优缺点

占内存小，可以实现高并发连接、处理响应快。

可以实现http服务器、虚拟主机、反向代理、负载均衡。

nginx配置简单

可以不暴露真实服务器IP地址

### nginx.conf 介绍

#### nginx.conf文件的结构

nginx的配置由特定的标识符(指令符)分为多个不同的模块。   
指令符分为**简单指令**和**块指令**。

* 简单指令格式：[name parameters;]
* 块指令格式：和简单指令格式有一样的结构，但其结束标识符不是分号，而是大括号{},块指令内部可以包含simple directives 和block directives, 可以称块指令为上下文(e.g. events, http, server, location)

conf文件中，所有不属于块指令的简单指令都属于main上下文的，http块指令属于main上下文，server块指令http上下文。

#### 配置静态访问

Web server很重要一部分工作就是提供静态页面的访问，例如images, html page。nginx可以通过不同的配置，根据request请求，从本地的目录提供不同的文件返回给客户端。   
打开安装目录下的nginx.conf文件，默认配置文件已经在http指令块中创建了一个空的server块，在nginx-1.8.0中的http块中已经创建了一个默认的server块。内容如下：

|  |
| --- |
| **server {**  **listen 80;**  **server\_name localhost;**  **location / {**  **root html;**  **index index.html index.htm;**  **}**  **error\_page 500 502 503 504 /50x.html;**  **location = /50x.html {**  **root html;**  **}**  **}** |

### nginx实现反向代理

#### 什么是反向代理?

反向代理（Reverse Proxy）方式是指以[代理服务器](https://baike.baidu.com/item/%E4%BB%A3%E7%90%86%E6%9C%8D%E5%8A%A1%E5%99%A8)来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。

启动一个Tomcat 127.0.0.1:8080

使用nginx反向代理 8080.itmayiedu.com 直接跳转到127.0.0.1:8080

#### Host文件新增

|  |
| --- |
| 127.0.0.1 8080.itmayiedu.com  127.0.0.1 b8081.itmayiedu.com |

#### nginx.conf 配置

配置信息:

|  |
| --- |
| server {  listen 80;  server\_name 8080.itmayiedu.com;  location / {  proxy\_pass http://127.0.0.1:8080;  index index.html index.htm;  }  }  server {  listen 80;  server\_name b8081.itmayiedu.com;  location / {  proxy\_pass http://127.0.0.1:8081;  index index.html index.htm;  }  } |

### nginx实现负载均衡

#### 什么是负载均衡

负载均衡 建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

负载均衡，英文名称为Load Balance，其意思就是分摊到多个操作单元上进行执行，例如Web服务器、FTP服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。



#### 负载均衡策略

|  |
| --- |
| **1、轮询（默认）** 每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。  upstream backserver {  server 192.168.0.14;  server 192.168.0.15;  }   **2、指定权重** 指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况。  upstream backserver {  server 192.168.0.14 weight=10;  server 192.168.0.15 weight=10;  }   **3、IP绑定 ip\_hash** 每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器，可以解决session的问题。  upstream backserver {  ip\_hash;  server 192.168.0.14:88;  server 192.168.0.15:80;  } |

#### 配置代码

|  |
| --- |
| upstream backserver {  server 127.0.0.1:8080;  server 127.0.0.1:8081;  }  server {  listen 80;  server\_name www.itmayiedu.com;  location / {  proxy\_pass http://backserver;  index index.html index.htm;  }  } |

#### 宕机轮训配置规则

|  |
| --- |
| server {  listen 80;  server\_name www.itmayiedu.com;  location / {  proxy\_pass http://backserver;  index index.html index.htm;  proxy\_connect\_timeout 1;  proxy\_send\_timeout 1;  proxy\_read\_timeout 1;  }    } |

### 负载均衡服务器有哪些？

LVS、Ngnix、Tengine（taobao 开发的 Nginx 升级版）、HAProxy（高可用、负载均衡）、Keepalived（故障转移，备机，linux 环境下的组件）

### nginx解决网站跨域问题

配置:

|  |
| --- |
| server {  listen 80;  server\_name www.itmayiedu.com;  location /A {  proxy\_pass http://a.a.com:81/A;  index index.html index.htm;  }  location /B {  proxy\_pass http://b.b.com:81/B;  index index.html index.htm;  }  } |

### nginx配置防盗链

|  |
| --- |
| location ~ .\*\.(jpg|jpeg|JPG|png|gif|icon)$ {  valid\_referers blocked http://www.itmayiedu.com www.itmayiedu.com;  if ($invalid\_referer) {  return 403;  }  } |

### nginx配置DDOS

#### 限制请求速度

设置Nginx、Nginx Plus的连接请求在一个真实用户请求的合理范围内。比如，如果你觉得一个正常用户每两秒可以请求一次登录页面，你就可以设置Nginx每两秒钟接收一个客户端IP的请求（大约等同于每分钟30个请求）。

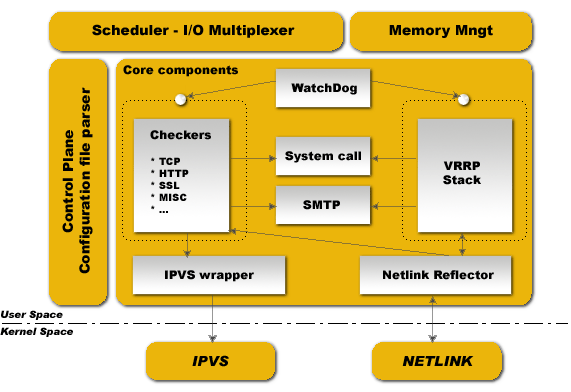
|  |
| --- |
| limit\_req\_zone $binary\_remote\_addr zone=one:10m rate=30r/m;  server {  ...  location /login.html {  limit\_req zone=one;  ...  }  } |

`limit\_req\_zone`命令设置了一个叫one的共享内存区来存储请求状态的特定键值，在上面的例子中是客户端IP($binary\_remote\_addr)。location块中的`limit\_req`通过引用one共享内存区来实现限制访问/login.html的目的。

#### 限制请求速度

设置Nginx、Nginx Plus的连接数在一个真实用户请求的合理范围内。比如，你可以设置每个客户端IP连接/store不可以超过10个。

### 什么是Keepalived

[Keepalived](http://www.keepalived.org/)是一个免费开源的，用C编写的类似于layer3, 4 & 7交换机制软件，具备我们平时说的第3层、第4层和第7层交换机的功能。主要提供**loadbalancing**（负载均衡）和 **high-availability**（高可用）功能，负载均衡实现需要依赖Linux的虚拟服务内核模块（ipvs），而高可用是通过VRRP协议实现多台机器之间的故障转移服务。   
   
上图是Keepalived的功能体系结构，大致分两层：用户空间（user space）和内核空间（kernel space）。   
**内核空间**：主要包括IPVS（IP虚拟服务器，用于实现网络服务的负载均衡）和NETLINK（提供高级路由及其他相关的网络功能）两个部份。   
**用户空间**：

* WatchDog：负载监控checkers和VRRP进程的状况
* VRRP Stack：负载负载均衡器之间的失败切换FailOver，如果只用一个负载均稀器，则VRRP不是必须的。
* Checkers：负责真实服务器的健康检查healthchecking，是keepalived最主要的功能。换言之，可以没有VRRP Stack，但健康检查healthchecking是一定要有的。
* IPVS wrapper：用户发送设定的规则到内核ipvs代码
* Netlink Reflector：用来设定vrrp的vip地址等。

Keepalived的所有功能是配置keepalived.conf文件来实现的。

### 集群情况下Session共享解决方案

#### nginx或者haproxy做的负载均衡)

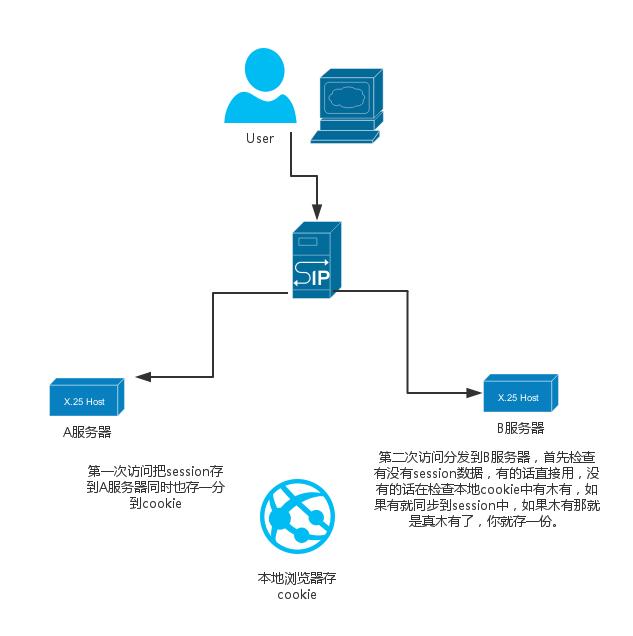
用Nginx 做的负载均衡可以添加ip\_hash这个配置，

用haproxy做的负载均衡可以用 balance source这个配置。

从而使同一个ip的请求发到同一台服务器。

#### 利用数据库同步session

#### 利用cookie同步session数据原理图如下



缺点:安全性差、http请求都需要带参数增加了带宽消耗

#### 使用Session集群存放Redis

使用spring-session框架，底层实现原理是重写httpsession

##### 引入maven依赖

|  |
| --- |
| **<!--spring boot 与redis应用基本环境配置 -->**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-redis</artifactId>**  **</dependency>**  **<!--spring session 与redis应用基本环境配置,需要开启redis后才可以使用，不然启动Spring boot会报错 -->**  **<dependency>**  **<groupId>org.springframework.session</groupId>**  **<artifactId>spring-session-data-redis</artifactId>**  **</dependency>** |

##### 创建SessionConfig

|  |
| --- |
| **import org.springframework.beans.factory.annotation.Value;**  **import org.springframework.context.annotation.Bean;**  **import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;**  **import org.springframework.session.data.redis.config.annotation.web.http.EnableRedisHttpSession;**  **//这个类用配置redis服务器的连接**  **//maxInactiveIntervalInSeconds为SpringSession的过期时间（单位：秒）**  **@EnableRedisHttpSession(maxInactiveIntervalInSeconds = 1800)**  **public class SessionConfig {**  **// 冒号后的值为没有配置文件时，制动装载的默认值**  **@Value("${redis.hostname:localhost}")**  **String HostName;**  **@Value("${redis.port:6379}")**  **int Port;**  **@Bean**  **public JedisConnectionFactory connectionFactory() {**  **JedisConnectionFactory connection = new JedisConnectionFactory();**  **connection.setPort(Port);**  **connection.setHostName(HostName);**  **return connection;**  **}**  **}** |

##### 初始化Session

|  |
| --- |
| **//初始化Session配置**  **public class SessionInitializer extends AbstractHttpSessionApplicationInitializer{**  **public SessionInitializer() {**  **super(SessionConfig.class);**  **}**  **}** |

##### 控制器层代码

|  |
| --- |
| import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpSession;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.boot.SpringApplication;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  @RestController  public class SessionController {  @Value("${server.port}")  private String PORT;  public static void main(String[] args) {  SpringApplication.run(SessionController.class, args);  }  @RequestMapping("/index")  public String index() {  return "index:" + PORT;  }  /\*\*  \*  \* @methodDesc: 功能描述:(往session存放值)  \* @author: 余胜军  \* @param: @param  \* httpSession  \* @param: @param  \* sessionKey  \* @param: @param  \* sessionValue  \* @param: @return  \* @createTime:2017年10月8日 下午3:55:26  \* @returnType:@param httpSession  \* @returnType:@param sessionKey  \* @returnType:@param sessionValue  \* @returnType:@return String  \* @copyright:上海每特教育科技有限公司  \* @QQ:644064779  \*/  @RequestMapping("/setSession")  public String setSession(HttpServletRequest request, String sessionKey, String sessionValue) {  HttpSession session = request.getSession(true);  session.setAttribute(sessionKey, sessionValue);  return "success,port:" + PORT;  }  /\*\*  \*  \* @methodDesc: 功能描述:(从Session获取值)  \* @author: 余胜军  \* @param: @param  \* httpSession  \* @param: @param  \* sessionKey  \* @param: @return  \* @createTime:2017年10月8日 下午3:55:47  \* @returnType:@param httpSession  \* @returnType:@param sessionKey  \* @returnType:@return String  \* @copyright:上海每特教育科技有限公司  \* @QQ:644064779  \*/  @RequestMapping("/getSession")  public String getSession(HttpServletRequest request, String sessionKey) {  HttpSession session =null;  try {  session = request.getSession(false);  } catch (Exception e) {  e.printStackTrace();  }  String value=null;  if(session!=null){  value = (String) session.getAttribute(sessionKey);  }  return "sessionValue:" + value + ",port:" + PORT;  }  } |

### 高并发解决方案

业务数据库  -》 数据水平分割(分区分表分库)、读写分离、SQL优化

业务应用 -》 逻辑代码优化(算法优化)、公共数据缓存

应用服务器 -》 反向静态代理、配置优化、负载均衡(apache分发，多tomcat实例)

系统环境 -》 JVM调优

页面优化 -》 减少页面连接数、页面尺寸瘦身

1、动态资源和静态资源分离；

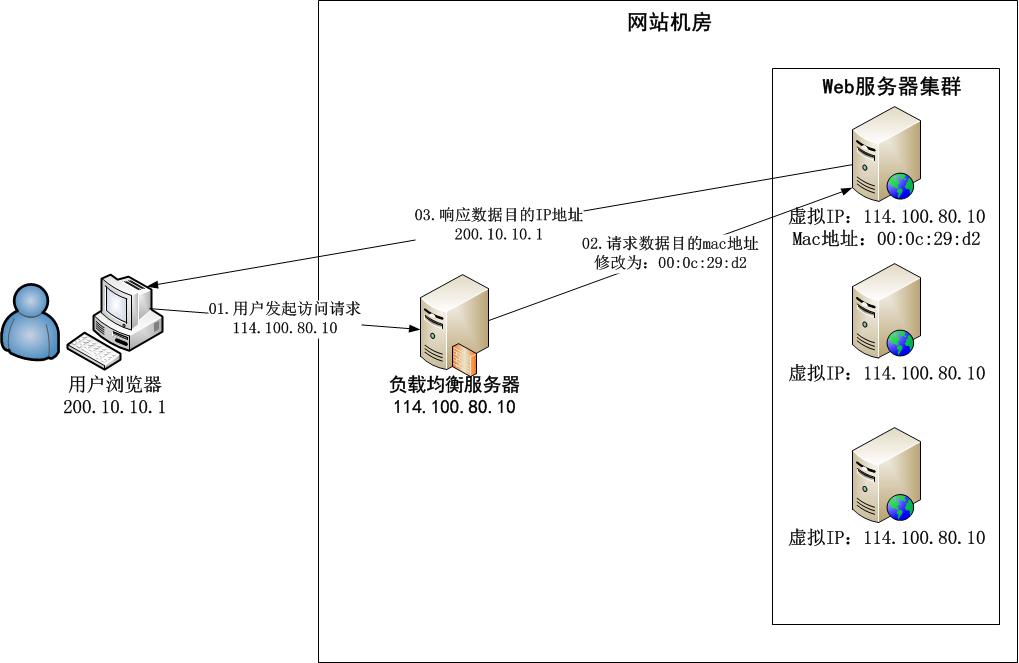
2、CDN；

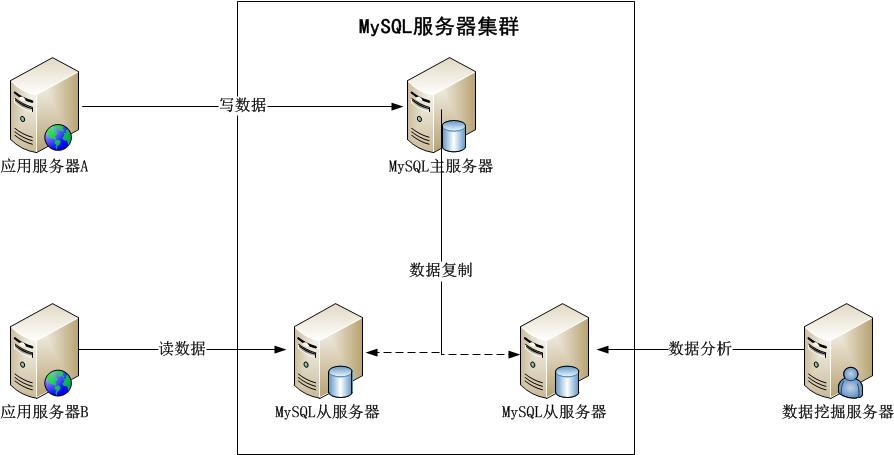
3、负载均衡；

4、分布式缓存；

5、数据库读写分离或数据切分（垂直或水平）；

6、服务分布式部署。





## 消息中间件

### 消息中间件产生的背景

在客户端与服务器进行通讯时.客户端调用后，必须等待服务对象完成处理返回结果才能继续执行。

**客户与服务器对象的生命周期紧密耦合,客户进程和服务对象进程都都必须正常运行;如果由于服务对象崩溃或者网络故障导致用户的请求不可达,客户会受到异常**

**点对点通信: 客户的一次调用只发送给某个单独的目标对象。**

(画图演示)

### 什么是消息中间件

面向消息的中间件(MessageOrlented MiddlewareMOM)较好的解决了以上问  
题。发送者将消息发送给消息服务器，消息服务器将消感存放在若千队列中，在合适  
的时候再将消息转发给接收者。

这种模式下，发送和接收是异步的，发送者无需等  
待; 二者的生命周期未必相同: 发送消息的时候接收者不一定运行，接收消息的时候  
发送者也不一定运行;一对多通信: 对于一个消息可以有多个接收者。

### JMS介绍

#### 什么是JMS？

JMS是java的消息服务，JMS的客户端之间可以通过JMS服务进行异步的消息传输。

#### 什么是消息模型

|  |
| --- |
| ○ Point-to-Point(P2P) --- 点对点  ○ Publish/Subscribe(Pub/Sub)--- 发布订阅 |

即点对点和发布订阅模型

##### P2P (点对点)

P2P

1. P2P模式图   
   
2. 涉及到的概念   
   1. 消息队列（Queue）
   2. 发送者(Sender)
   3. 接收者(Receiver)
   4. 每个消息都被发送到一个特定的队列，接收者从队列中获取消息。队列保留着消息，直到他们被消费或超时。
3. P2P的特点
   1. 每个消息只有一个消费者（Consumer）(即一旦被消费，消息就不再在消息队列中)
   2. 发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列
   3. 接收者在成功接收消息之后需向队列应答成功

如果你希望发送的每个消息都应该被成功处理的话，那么你需要P2P模式。

应用场景

A用户与B用户发送消息

##### Pub/Sub (发布与订阅)

Pub/Sub模式图   


涉及到的概念 

主题（Topic）

发布者（Publisher）

订阅者（Subscriber）   
客户端将消息发送到主题。多个发布者将消息发送到Topic,系统将这些消息传递给多个订阅者。

Pub/Sub的特点

每个消息可以有多个消费者

发布者和订阅者之间有时间上的依赖性。针对某个主题（Topic）的订阅者，它必须创建一个订阅者之后，才能消费发布者的消息，而且为了消费消息，订阅者必须保持运行的状态。

为了缓和这样严格的时间相关性，JMS允许订阅者创建一个可持久化的订阅。这样，即使订阅者没有被激活（运行），它也能接收到发布者的消息。

如果你希望发送的消息可以不被做任何处理、或者被一个消息者处理、或者可以被多个消费者处理的话，那么可以采用Pub/Sub模型

消息的消费   
在JMS中，消息的产生和消息是异步的。对于消费来说，JMS的消息者可以通过两种方式来消费消息。   
○ 同步   
订阅者或接收者调用receive方法来接收消息，receive方法在能够接收到消息之前（或超时之前）将一直阻塞   
○ 异步   
订阅者或接收者可以注册为一个消息监听器。当消息到达之后，系统自动调用监听器的onMessage方法。

应用场景:

用户注册、订单修改库存、日志存储

画图演示







### MQ产品的分类

#### RabbitMQ

是使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正是如此，使的它变的非常重量级，更适合于企业级的开发。同时实现了一个经纪人(Broker)构架，这意味着消息在发送给客户端时先在中心队列排队。对路由(Routing)，负载均衡(Load balance)或者数据持久化都有很好的支持。

#### Redis

是一个Key-Value的NoSQL数据库，开发维护很活跃，虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 入队 | | | | 出队 | | | |
|  | 128B | 512B | 1K | 10K | 128B | 512B | 1K | 10K |
| Redis | 16088 | 15961 | 17094 | 25 | 15955 | 20449 | 18098 | 9355 |
| RabbitMQ | 10627 | 9916 | 9370 | 2366 | 3219 | 3174 | 2982 | 1588 |

#### ZeroMQ

号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZMQ能够实现RabbitMQ不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这MQ能够应用成功的挑战。ZeroMQ具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演了这个服务角色。你只需要简单的引用ZeroMQ程序库，可以使用NuGet安装，然后你就可以愉快的在应用程序之间发送消息了。但是ZeroMQ仅提供非持久性的队列，也就是说如果down机，数据将会丢失。其中，Twitter的Storm中使用ZeroMQ作为数据流的传输。

#### ActiveMQ

是Apache下的一个子项目。 类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。RabbitMQ、ZeroMQ、ActiveMQ均支持常用的多种语言客户端 C++、Java、.Net,、Python、 Php、 Ruby等。

#### Jafka/Kafka

Kafka是Apache下的一个子项目，是一个高性能跨语言分布式Publish/Subscribe消息队列系统，而Jafka是在Kafka之上孵化而来的，即Kafka的一个升级版。具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现复杂均衡；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka通过Hadoop的并行加载机制来统一了在线和离线的消息处理，这一点也是本课题所研究系统所看重的。Apache Kafka相对于ActiveMQ是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

其他一些队列列表HornetQ、Apache Qpid、Sparrow、Starling、Kestrel、Beanstalkd、Amazon SQS就不再一一分析。

### MQ怎么保证消息幂等问题

1. 发送端MQ-client 将消息发送给服务端MQ-server
2. 服务端MQ-server将消息落地
3. 服务端MQ-server 回ACK(表示确认) 2.如果3丢失 发送端在超时后，又会发送一遍，此时重发是MQ-client发起的，消息处理的是MQ-server 为了避免2 重复落地，对每条MQ消息系统内部需要生成一个inner-msg-id,作为去重和幂等的依据，这个内部消息ID 的特点是

在分布式环境中，MQ通讯产生网络延迟，重试补偿中，会造成MQ重复消费。

解决办法:

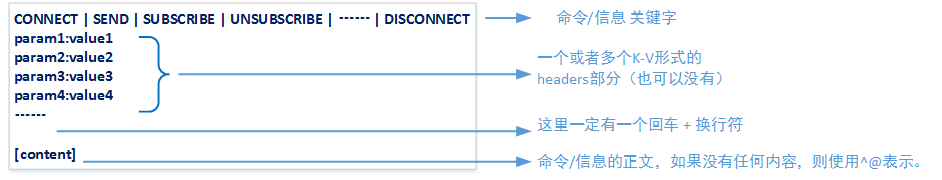
1. 使用日志+msg-id保存报文信息，作为去重和幂等的依据。
2. 消费端代码抛出异常，不需要重试补偿，使用日志记录报文，下次发版本解决。

### MQ有哪些协议

Stomp、XMPP

Stomp协议，英文全名Streaming Text Orientated Message Protocol，中文名称为 ‘流文本定向消息协议’。是一种以纯文本为载体的协议（以文本为载体的意思是它的消息格式规范中没有类似XMPP协议那样的xml格式要求，你可以将它看作‘半结构化数据’）。目前Stomp协议有两个版本：V1.1和V1.2。

一个标准的Stomp协议包括以下部分：命令/信息关键字、头信息、文本内容。如下图所示：



以下为一段简单的协议信息示例：

CONNECT

accept-version:1.2

someparam1:value1

someparam2:value2

上面的示例中，我们使用了Stomp协议的CONNECT命令，它的意思为连接到Stomp代理端，并且携带了要求代理端的版本信息和两个自定义的K-V信息（请注意’^@’符号，STOMP协议中用它来表示NULL）。

XMPP基于XML，用于IM系统的开发。国内比较流行的XMPP服务器叫做Openfire，它使用MINA作为下层的网络IO框架（不是MINA2是MINA1）；国外用的比较多的XMPP服务器叫做Tigase，它的官网号称单节点可以支撑50万用户在线，集群可以支持100万用户在线：（<http://projects.tigase.org/>）

# 微服务与分布式

## 什么是RPC远程调用？

RPC 的全称是 Remote Procedure Call 是一种进程间通信方式。  
它允许程序调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数，而**不用程序员显式编码这个远程调用的细节**。即无论是调用本地接口/服务的还是远程的接口/服务，本质上编写的调用代码基本相同。  
比如两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数或者方法，由于不在一个内存空间，不能直接调用，这时候需要通过就可以应用RPC框架的实现来解决

## 什么是SOA？与SOAP区别是什么？

SOA是一种面向服务架构，是将相同业务逻辑抽取出来组成单独服务。

SOAP是WebService面向服务协议， 采用xml,因为比较中，现在不是特别流行。

## 什么是微服务架构

微服务架构师一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相

协调、互相配合没用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的同学机制互相沟通(通畅采用Http+restful API)，每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生成环境、类生存环境等。另外，应尽量避免同一的、集中式服务管理机制。

## 微服务与SOA区别

|  |  |
| --- | --- |
| SOA实现 | 微服务架构实现 |
| 企业级，自顶向下开展实施 | 团队级，自定向上开展实施 |
| 服务由多个子系统组成 | 一个系统被拆分成多个服务 |
| 集成式服务(esb、ws、soap) | 集成方式简单(http、rest、json) |

## RPC远程调用有哪些框架？

SpringCloud、Dubbo、Dubbox、Hessian、HttpClient、thrift等。

## 什么是SpringCloud

SpringCloud 为开发人员提供了快速构建分布式系统的一些工具，包括配置管理、服务发现(Eureka)、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等。它运行环境简单，可以在开发人员的电脑上跑。另外说明spring cloud是基于Springboot的，所以需要开发中对Springboot有一定的了解，如果不了解的话可以看蚂蚁课堂SpringBoot课程。

SpringCloud使用Eureka作为注册中心，使用rest+ribbon或者feign，断路器Hystrix、zuul接口网关等。

## 什么是Dubbo？

**Duubbo是一个RPC远程调用框架， 分布式服务治理框架**

**什么是Dubbo服务治理？**

**服务与服务之间会有很多个Url、依赖关系、负载均衡、容错、自动注册服务。**

## Dubbo有哪些协议？

**默认用的dubbo协议、Http、RMI、**Hessian

## Dubbo整个架构流程

**分为四大模块**

**生产者、消费者、注册中心、监控中心**

**生产者：提供服务**

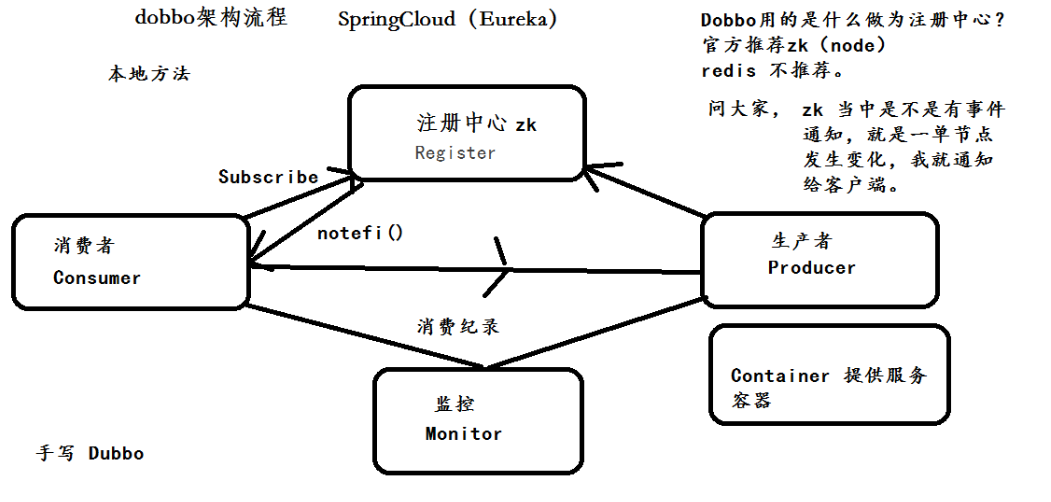
**消费者: 调用服务**

**注册中心:注册信息(redis、zk)**

**监控中心:调用次数、关系依赖等。**

**首先生产者将服务注册到注册中心（zk），使用zk持久节点进行存储，消费订阅zk节点，一旦有节点变更，zk通过事件通知传递给消费者，消费可以调用生产者服务。**

**服务与服务之间进行调用，都会在监控中心中，存储一个记录。**



## Dubbox与Dubbo区别？

**Dubox使用http协议+rest风格传入json或者xml格式进行远程调用。**

**Dubbo使用Dubbo协议。**

## SpringCloud与Dubbo区别？

### 相同点:

**dubbo与springcloud都可以实现RPC远程调用。**

**dubbo与springcloud都可以使用分布式、微服务场景下。**

### 区别:

**dubbo有比较强的背景,在国内有一定影响力。**

**dubbo使用zk或redis作为作为注册中心**

**springcloud使用eureka作为注册中心**

**dubbo支持多种协议，默认使用dubbo协议。**

**Springcloud只能支持http协议。**

**Springcloud是一套完整的微服务解决方案。**

**Dubbo目前已经停止更新,SpringCloud更新速度快。**

## 什么是Zookeeper

**Zookeeper是一个工具，可以实现集群中的分布式协调服务。**

**所谓的分布式协调服务，就是在集群的节点中进行可靠的消息传递，来协调集群的工作。**

**Zookeeper之所以能够实现分布式协调服务，靠的就是它能够保证分布式数据一致性。**

**所谓的分布式数据一致性，指的就是可以在集群中保证数据传递的一致性。**

**Zookeeper能够提供的分布式协调服务包括：数据发布订阅、负载均衡、命名服务、分布式协调/通知、集群管理、分布式锁、分布式队列等功能**

## Zookeeper特点

**Zookeeper工作在集群中，对集群提供分布式协调服务，它提供的分布式协调服务具有如下的特点：**

**顺序一致性**

**从同一个客户端发起的事务请求，最终将会严格按照其发起顺序被应用到zookeeper中**

**原子性**

**所有事物请求的处理结果在整个集群中所有机器上的应用情况是一致的，即，要么整个集群中所有机器都成功应用了某一事务，要么都没有应用，一定不会出现集群中部分机器应用了改事务，另外一部分没有应用的情况。**

**单一视图**

**无论客户端连接的是哪个zookeeper服务器，其看到的服务端数据模型都是一致的。**

**可靠性**

**一旦服务端成功的应用了一个事务，并完成对客户端的响应，那么该事务所引起的服务端状态变更将会一直保留下来，除非有另一个事务又对其进行了改变。**

**实时性**

**zookeeper并不是一种强一致性，只能保证顺序一致性和最终一致性，只能称为达到了伪实时性。**

## zookeeper的数据模型

**zookeepei中可以保存数据，正是利用zookeeper可以保存数据这一特点，我们的集群通过在zookeeper里存取数据来进行消息的传递。**

**zookeeper中保存数据的结构非常类似于文件系统。都是由节点组成的树形结构。不同的是文件系统是由文件夹和文件来组成的树，而zookeeper中是由ZNODE来组成的树。**

**每一个ZNODE里都可以存放一段数据，ZNODE下还可以挂载零个或多个子ZNODE节点，从而组成一个树形结构。**

## Zookeeper应用场景

数据发布订阅

负载均衡

命名服务

分布式协调

集群管理

配置管理

分布式队列

**分布式锁**

## 什么是分布式锁

简单的理解就是：分布式锁是一个在很多环境中非常有用的原语，它是不同的系统或是同一个系统的不同主机之间互斥操作共享资源的有效方法。

## Zookeeper实现分布式锁

分布式锁使用zk，在zk上创建一个临时节点，使用临时节点作为锁，因为节点不允许重复。

如果能创建节点成功，生成订单号，如果创建节点失败，就等待。临时节点zk关闭，释放锁，其他节点就可以重新生成订单号。

## Redis分布式锁思考

一般的锁只能针对单机下同一进程的多个线程，或单机的多个进程。多机情况下，对同一个资源访问，需要对每台机器的访问进程或线程加锁，这便是分布式锁。分布式锁可以利用多机的共享缓存（例如redis）实现。redis的命令文档[1]，实现及分析参考文档[2]。

利用redis的get、setnx、getset、del四个命令可以实现基于redis的分布式锁：

get key：表示从redis中读取一个key的value，如果key没有对应的value，返回nil，如果存储的值不是字符串类型，返回错误

setnx key value：表示往redis中存储一个键值对，但只有当key不存在时才成功，返回1；否则失败，返回0，不改变key的value

getset key：将给定 key 的值设为 value ，并返回 key 的旧值(old value)。当旧值不存在时返回nil，当旧值不为字符串类型，返回错误

del key：表示删除key，当key不存在时不做操作，返回删除的key数量

关于加锁思考，循环中：   
0、setnx的value是当前机器时间+预估运算时间作为锁的失效时间。这是为了防止获得锁的线程挂掉而无法释放锁而导致死锁。   
0.1、返回1，证明已经获得锁，返回啦   
0.2、返回0，获得锁失败，需要检查锁超时时间   
1、get 获取到锁，利用失效时间判断锁是否失效。   
1.1、取锁超时时间的时刻可能锁被删除释放，此时并没有拿到锁，应该重新循环加锁逻辑。   
2、取锁超时时间成功   
2.1、锁没有超时，休眠一下，重新循环加锁   
2.2、锁超时，但此时不能直接释放锁删除。因为此时可能多个线程都读到该锁超时，如果直接删除锁，所有线程都可能删除上一个删除锁又新上的锁，会有多个线程进入临界区，产生竞争状态。   
3、此时采用乐观锁的思想，用getset再次获取锁的旧超时时间。   
3.1、如果此时获得锁旧超时时间成功   
3.1.1、等于上一次获得的锁超时时间，证明两次操作过程中没有别人动过这个锁，此时已经获得锁   
3.1.2、不等于上一次获得的锁超时时间，说明有人先动过锁，获取锁失败。虽然修改了别人的过期时间，但因为冲突的线程相差时间极短，所以修改后的过期时间并无大碍。此处依赖所有机器的时间一致。   
3.2、如果此时获得锁旧超时时间失败，证明当前线程是第一个在锁失效后又加上锁的线程，所以也获得锁   
4、其他情况都没有获得锁，循环setnx吧

关于解锁的思考：   
在锁的时候，如果锁住了，回传超时时间，作为解锁时候的凭证，解锁时传入锁的键值和凭证。我思考的解锁时候有两种写法：   
1、解锁前get一下键值的value，判断是不是和自己的凭证一样。但这样存在一些问题：

get时返回nil的可能，此时表示有别的线程拿到锁并用完释放

get返回非nil，但是不等于自身凭证。由于有getset那一步，当两个竞争线程都在这个过程中时，存在持有锁的线程凭证不等于value，而是value是稍慢那一步线程设置的value。

2、解锁前用凭证判断锁是否已经超时，如果没有超时，直接删除；如果超时，等着锁自动过期就好，免得误删别人的锁。但这种写法同样存在问题，由于线程调度的不确定性，判断到删除之间可能过去很久，并不是绝对意义上的正确解锁。

|  |
| --- |
| public class RedisLock {  private static final Logger logger = LoggerFactory.getLogger(RedisLock.class);  //显然jedis还需要自己配置来初始化  private Jedis jedis = new Jedis();  //默认锁住15秒，尽力规避锁时间太短导致的错误释放  private static final long DEFAULT\_LOCK\_TIME = 15 \* 1000;  //尝试锁住一个lock，设置尝试锁住的次数和超时时间（毫秒）,默认最短15秒  //成功时返回这把锁的key，解锁时需要凭借锁的lock和key  //失败时返回空字符串  public String lock(String lock, int retryCount, long timeout) {  Preconditions.checkArgument(retryCount > 0 && timeout > 0, "retry count <= 0 or timeout <= 0 !");  Preconditions.checkArgument(retryCount < Integer.MAX\_VALUE && timeout < Long.MAX\_VALUE - DEFAULT\_LOCK\_TIME,  "retry count is too big or timeout is too big!");  String $lock = Preconditions.checkNotNull(lock) + "\_redis\_lock";  long $timeout = timeout + DEFAULT\_LOCK\_TIME;  String ret = null;  //重试一定次数，还是拿不到，就放弃  try {  long i, status;  for (i = 0, status = 0; status == 0 && i < retryCount; ++i) {  //尝试加锁，并设置超时时间为当前机器时间+超时时间  if ((status = jedis.setnx($lock, ret = Long.toString(System.currentTimeMillis() + $timeout))) == 0) {  //获取锁失败，查看锁是否超时  String time = jedis.get($lock);  //在加锁和检查之间，锁被删除了，尝试重新加锁  if (time == null) {  continue;  }  //锁的超时时间戳小于当前时间，证明锁已经超时  if (Long.parseLong(time) < System.currentTimeMillis()) {  String oldTime = jedis.getSet($lock, Long.toString(System.currentTimeMillis() + $timeout));  if (oldTime == null || oldTime.equals(time)) {  //拿到锁了，跳出循环  break;  }  }  try {  TimeUnit.MILLISECONDS.sleep(1L);  } catch (InterruptedException e) {  logger.error("lock key:{} sleep failed!", lock);  }  }  }  if (i == retryCount && status == 0) {  logger.info("lock key:{} failed!", lock);  return "";  }  //给锁加上过期时间  jedis.pexpire($lock, $timeout);  logger.info("lock key:{} succsee!", lock);  return ret;  } catch (Exception e) {  logger.error("redis lock key:{} failed! cached exception: ", lock, e);  return "";  }  }  //释放lock的锁，需要传入lock和key  //尽力确保删除属于自己的锁，但是不保证做得到  public void releaseLock(String lock, String key) {  String $lock = Preconditions.checkNotNull(lock) + "\_redis\_lock";  Preconditions.checkNotNull(key);  try {  long timeout = Long.parseLong(key);  //锁还没有超时，锁还属于自己可以直接删除  //但由于线程运行的不确定性，其实不能完全保证删除时锁还属于自己  //真正执行删除操作时，距离上语句判断可能过了很久  if (timeout <= System.currentTimeMillis()) {  jedis.del($lock);  logger.info("release lock:{} with key:{} success!", lock, key);  } else {  logger.info("lock:{} with key:{} timeout! wait to expire", lock, key);  }  } catch (Exception e) {  logger.error("redis release {} with key:{} failed! cached exception: ", lock, key, e);  }  }  } |

## Zookeeper与 Redis实现分布式锁的区别

### 基于缓存实现分布式锁

锁没有失效事件,容易死锁

非阻塞式

不可重入

### 基于Zookeeper实现分布式锁

实现相对简单

可靠性高

性能较好

## Java实现定时任务有哪些方式

### Thread

|  |
| --- |
| **public class Demo01 {**  **static long *count* = 0;**  **public static void main(String[] args) {**  **Runnable runnable = new Runnable() {**  **@Override**  **public void run() {**  **while (true) {**  **try {**  **Thread.*sleep*(1000);**  ***count*++;**  **System.*out*.println(*count*);**  **} catch (Exception e) {**  **// TODO: handle exception**  **}**  **}**  **}**  **};**  **Thread thread = new Thread(runnable);**  **thread.start();**  **}**  **}** |

### TimerTask

|  |
| --- |
| **/\*\***  **\* 使用TimerTask类实现定时任务**  **\*/**  **public class Demo02 {**  **static long *count* = 0;**  **public static void main(String[] args) {**  **TimerTask timerTask = new TimerTask() {**  **@Override**  **public void run() {**  ***count*++;**  **System.*out*.println(*count*);**  **}**  **};**  **Timer timer = new Timer();**  **// 天数**  **long delay = 0;**  **// 秒数**  **long period = 1000;**  **timer.scheduleAtFixedRate(timerTask, delay, period);**  **}**  **}** |

### ScheduledExecutorService

使用ScheduledExecutorService是从Java

JavaSE5的java.util.concurrent里，做为并发工具类被引进的，这是最理想的定时任务实现方式。

|  |
| --- |
| **public class Demo003 {**  **public static void main(String[] args) {**  **Runnable runnable = new Runnable() {**  **public void run() {**  **// task to run goes here**  **System.*out*.println("Hello !!");**  **}**  **};**  **ScheduledExecutorService service = Executors.*newSingleThreadScheduledExecutor*();**  **// 第二个参数为首次执行的延时时间，第三个参数为定时执行的间隔时间**  **service.scheduleAtFixedRate(runnable, 1, 1, TimeUnit.*SECONDS*);**  **}**  **}** |

### Quartz

#### 创建一个quartz\_demo项目

#### 引入maven依赖

|  |
| --- |
| **<dependencies>**  **<!-- quartz -->**  **<dependency>**  **<groupId>org.quartz-scheduler</groupId>**  **<artifactId>quartz</artifactId>**  **<version>2.2.1</version>**  **</dependency>**  **<dependency>**  **<groupId>org.quartz-scheduler</groupId>**  **<artifactId>quartz-jobs</artifactId>**  **<version>2.2.1</version>**  **</dependency>**  **</dependencies>** |

#### 任务调度类

|  |
| --- |
| **public class MyJob implements Job {**  **public void execute(JobExecutionContext context) throws JobExecutionException {**  **System.*out*.println("quartz MyJob date:" + new Date().getTime());**  **}**  **}** |

#### 启动类

|  |
| --- |
| **//1.创建Scheduler的工厂**  **SchedulerFactory sf = new StdSchedulerFactory();**  **//2.从工厂中获取调度器实例**  **Scheduler scheduler = sf.getScheduler();**  **//3.创建JobDetail**  **JobDetail jb = JobBuilder.*newJob*(MyJob.class)**  **.withDescription("this is a ram job") //job的描述**  **.withIdentity("ramJob", "ramGroup") //job 的name和group**  **.build();**  **//任务运行的时间，SimpleSchedle类型触发器有效**  **long time= System.*currentTimeMillis*() + 3\*1000L; //3秒后启动任务**  **Date statTime = new Date(time);**  **//4.创建Trigger**  **//使用SimpleScheduleBuilder或者CronScheduleBuilder**  **Trigger t = TriggerBuilder.*newTrigger*()**  **.withDescription("")**  **.withIdentity("ramTrigger", "ramTriggerGroup")**  **//.withSchedule(SimpleScheduleBuilder.simpleSchedule())**  **.startAt(statTime) //默认当前时间启动**  **.withSchedule(CronScheduleBuilder.*cronSchedule*("0/2 \* \* \* \* ?")) //两秒执行一次**  **.build();**  **//5.注册任务和定时器**  **scheduler.scheduleJob(jb, t);**  **//6.启动 调度器**  **scheduler.start();** |

## 分布式情况下定时任务会出现哪些问题？

分布式集群的情况下，怎么保证定时任务不被重复执行

### 分布式定时任务解决方案

①使用zookeeper实现分布式锁 缺点(需要创建临时节点、和事件通知不易于扩展)

②使用配置文件做一个开关 缺点发布后，需要重启

③数据库唯一约束，缺点效率低

④使用分布式任务调度平台

XXLJOB

## 分布式事物解决方案

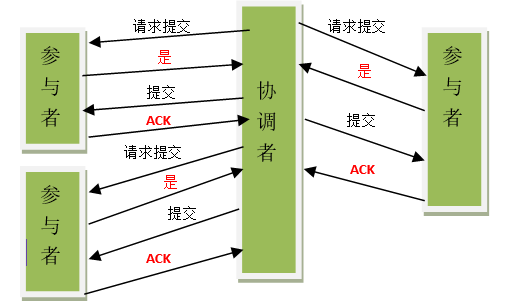
### 全局事物

使用全局事物两段提交协议，遵循XA协议规范，使用开源框架jta+automatic 。

什么是两段提交协议：在第一阶段，所有参与全局事物的节点都开始准备,告诉事物管理器

它们准备好了，在是第二阶段，事物管理器告诉资源执行ROLLBACK还是Commit，只要任何一方为ROLLBACK ，则直接回滚。

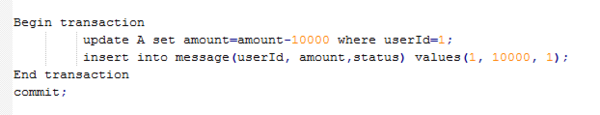
参考案例：springboot集成automatic+jta



### 本地消息表

这种实现方式的思路，其实是源于ebay，后来通过支付宝等公司的布道，在业内广泛使用。其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。如果不考虑性能及设计优雅，借助关系型数据库中的表即可实现。

举个经典的跨行转账的例子来描述。

第一步，伪代码如下，扣款1W，通过本地事务保证了凭证消息插入到消息表中。   


第二步，通知对方银行账户上加1W了。那问题来了，如何通知到对方呢？

通常采用两种方式：

**1. 采用时效性高的MQ，由对方订阅消息并监听，有消息时自动触发事件   
2. 采用定时轮询扫描的方式，去检查消息表的数据。**

两种方式其实各有利弊，仅仅依靠MQ，可能会出现通知失败的问题。而过于频繁的定时轮询，效率也不是最佳的（90%是无用功）。所以，我们一般会把两种方式结合起来使用。

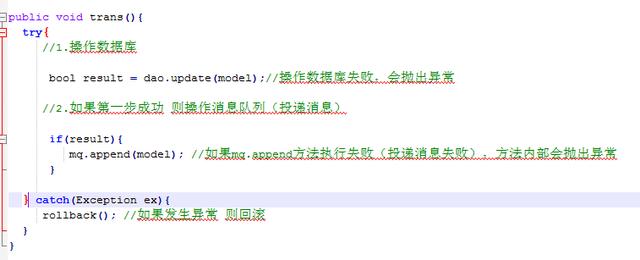
解决了通知的问题，又有新的问题了。万一这消息有重复被消费，往用户帐号上多加了钱，那岂不是后果很严重？

仔细思考，其实我们可以消息消费方，也通过一个“消费状态表”来记录消费状态。在执行“加款”操作之前，检测下该消息（提供标识）是否已经消费过，消费完成后，通过本地事务控制来更新这个“消费状态表”。这样子就避免重复消费的问题。

总结：上诉的方式是一种非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。但是，关系型数据库的吞吐量和性能方面存在瓶颈，频繁的读写消息会给数据库造成压力。所以，在真正的高并发场景下，该方案也会有瓶颈和限制的。

### MQ(非事物)

通常情况下，在使用非事务消息支持的MQ产品时，我们很难将业务操作与对MQ的操作放在一个本地事务域中管理。通俗点描述，还是以上述提到的“跨行转账”为例，我们很难保证在扣款完成之后对MQ投递消息的操作就一定能成功。这样一致性似乎很难保证。   
先从消息生产者这端来分析，请看伪代码：



根据上述代码及注释，我们来分析下可能的情况：

1. 操作数据库成功，向MQ中投递消息也成功，皆大欢喜   
2. 操作数据库失败，不会向MQ中投递消息了   
3. 操作数据库成功，但是向MQ中投递消息时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚

从上面分析的几种情况来看，貌似问题都不大的。那么我们来分析下消费者端面临的问题：

1. 消息出列后，消费者对应的业务操作要执行成功。如果业务执行失败，消息不能失效或者丢失。需要保证消息与业务操作一致   
2. 尽量避免消息重复消费。如果重复消费，也不能因此影响业务结果

如何保证消息与业务操作一致，不丢失？

主流的MQ产品都具有持久化消息的功能。如果消费者宕机或者消费失败，都可以执行重试机制的（有些MQ可以自定义重试次数）。

如何避免消息被重复消费造成的问题？

1. 保证消费者调用业务的服务接口的幂等性   
2. 通过消费日志或者类似状态表来记录消费状态，便于判断（建议在业务上自行实现，而不依赖MQ产品提供该特性）

总结：这种方式比较常见，性能和吞吐量是优于使用关系型数据库消息表的方案。如果MQ自身和业务都具有高可用性，理论上是可以满足大部分的业务场景的。不过在没有充分测试的情况下，不建议在交易业务中直接使用。

### MQ（事务消息）

### 其他补偿方式

做过支付宝交易接口的同学都知道，我们一般会在支付宝的回调页面和接口里，解密参数，然后调用系统中更新交易状态相关的服务，将订单更新为付款成功。同时，只有当我们回调页面中输出了success字样或者标识业务处理成功相应状态码时，支付宝才会停止回调请求。否则，支付宝会每间隔一段时间后，再向客户方发起回调请求，直到输出成功标识为止。   
其实这就是一个很典型的补偿例子，跟一些MQ重试补偿机制很类似。

一般成熟的系统中，对于级别较高的服务和接口，整体的可用性通常都会很高。如果有些业务由于瞬时的网络故障或调用超时等问题，那么这种重试机制其实是非常有效的。

当然，考虑个比较极端的场景，假如系统自身有bug或者程序逻辑有问题，那么重试1W次那也是无济于事的。那岂不是就发生了“明明已经付款，却显示未付款不发货”类似的悲剧？

其实为了交易系统更可靠，我们一般会在类似交易这种高级别的服务代码中，加入详细日志记录的，一旦系统内部引发类似致命异常，会有邮件通知。同时，后台会有定时任务扫描和分析此类日志，检查出这种特殊的情况，会尝试通过程序来补偿并邮件通知相关人员。

在某些特殊的情况下，还会有“人工补偿”的，这也是最后一道屏障。

### 补充资料

#### 什么是两段提交协议

##### 一、协议概述

　两阶段提交协议（two phase commit protocol，2PC）可以保证数据的强一致性，许多分布式关系型数据管理系统采用此协议来完成分布式事务。它是协调所有分布式原子事务参与者，并决定提交或取消（回滚）的分布式算法。同时也是解决一致性问题的一致性算法。该算法能够解决很多的临时性系统故障（包括进程、网络节点、通信等故障），被广泛地使用。但是，它并不能够通过配置来解决所有的故障，在某些情况下它还需要人为的参与才能解决问题。参与者为了能够从故障中恢复，它们都使用日志来记录协议的状态，虽然使用日志降低了性能但是节点能够从故障中恢复。

　　在两阶段提交协议中，系统一般包含两类机器（或节点）：  
　　**①**协调者coordinator，通常一个系统中只有一个；  
　　**②**事务参与者 participants，cohorts或workers，一般包含多个；  
　　在数据存储系统中可以理解为数据副本的个数，协议中假设：  
　　**①**每个节点都会记录写前日志并持久性存储，即使节点发生故障日志也不会丢失；  
　　**②**节点不会发生永久性故障而且任意两个节点都可以互相通信；

　　当事务的最后一步完成之后，协调器执行协议，参与者根据本地事务，能够成功完成回复同意提交事务或者回滚事务。

##### 二、执行过程

　　顾名思义，两阶段提交协议由两个阶段组成。在正常的执行下，这两个阶段的执行过程如下所述：

**（1）阶段1：请求阶段（commit-request phase，或称表决阶段，voting phase）**

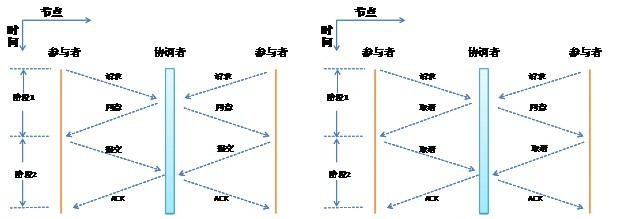
　　在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地作业执行成功）或取消（本地作业执行故障）。

**（2）阶段2：提交阶段（commit phase）**

　　在该阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消。当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。参与者在接收到协调者发来的消息后将执行响应的操作。

　　注意  两阶段提交协议与两阶段锁协议不同，两阶段锁协议为一致性控制协议。

（3）该协议的执行过程可以通过下图来描述：



               （a）成功                                   （b）失败

##### 三、协议的特点

　　两阶段提交协议最大的劣势是其通过阻塞完成的协议，在节点等待消息的时候处于阻塞状态，节点中其他进程则需要等待阻塞进程释放资源才能使用。如果协调器发生了故障，那么参与者将无法完成事务则一直等待下去。以下情况可能会导致节点发生永久阻塞：

　　（1）如果参与者发送同意提交消息给协调者，进程将阻塞直至收到协调器的提交或回滚的消息。如果协调器发生永久故障，参与者将一直等待，这里可以采用备份的协调器，所有参与者将回复发给备份协调器，由它承担协调器的功能。

　　（2）如果协调器发送“请求提交”消息给参与者，它将被阻塞直到所有参与者回复了，如果某个参与者发生永久故障，那么协调器也不会一直阻塞，因为协调器在某一时间内还未收到某参与者的消息，那么它将通知其他参与者回滚事务。

　　同时两阶段提交协议没有容错机制，一个节点发生故障整个事务都要回滚，代价比较大。

##### 四、工作过程

　　下面我们通过一个例子来说明两阶段提交协议的工作过程：

　　A组织B、C和D三个人去爬长城：如果所有人都同意去爬长城，那么活动将举行；如果有一人不同意去爬长城，那么活动将取消。用2PC算法解决该问题的过程如下：

　　首先A将成为该活动的协调者，B、C和D将成为该活动的参与者。

**（1）阶段1：**

　　①A发邮件给B、C和D，提出下周三去爬山，问是否同意。那么此时A需要等待B、C和D的邮件。

　　②B、C和D分别查看自己的日程安排表。B、C发现自己在当日没有活动安排，则发邮件告诉A它们同意下周三去爬长城。由于某种原因， D白天没有查看邮 件。那么此时A、B和C均需要等待。到晚上的时候，D发现了A的邮件，然后查看日程安排，发现周三当天已经有别的安排，那么D回复A说活动取消吧。

**（2）阶段2：**

　　①此时A收到了所有活动参与者的邮件，并且A发现D下周三不能去爬山。那么A将发邮件通知B、C和D，下周三爬长城活动取消。

　　②此时B、C回复A“太可惜了”，D回复A“不好意思”。至此该事务终止。

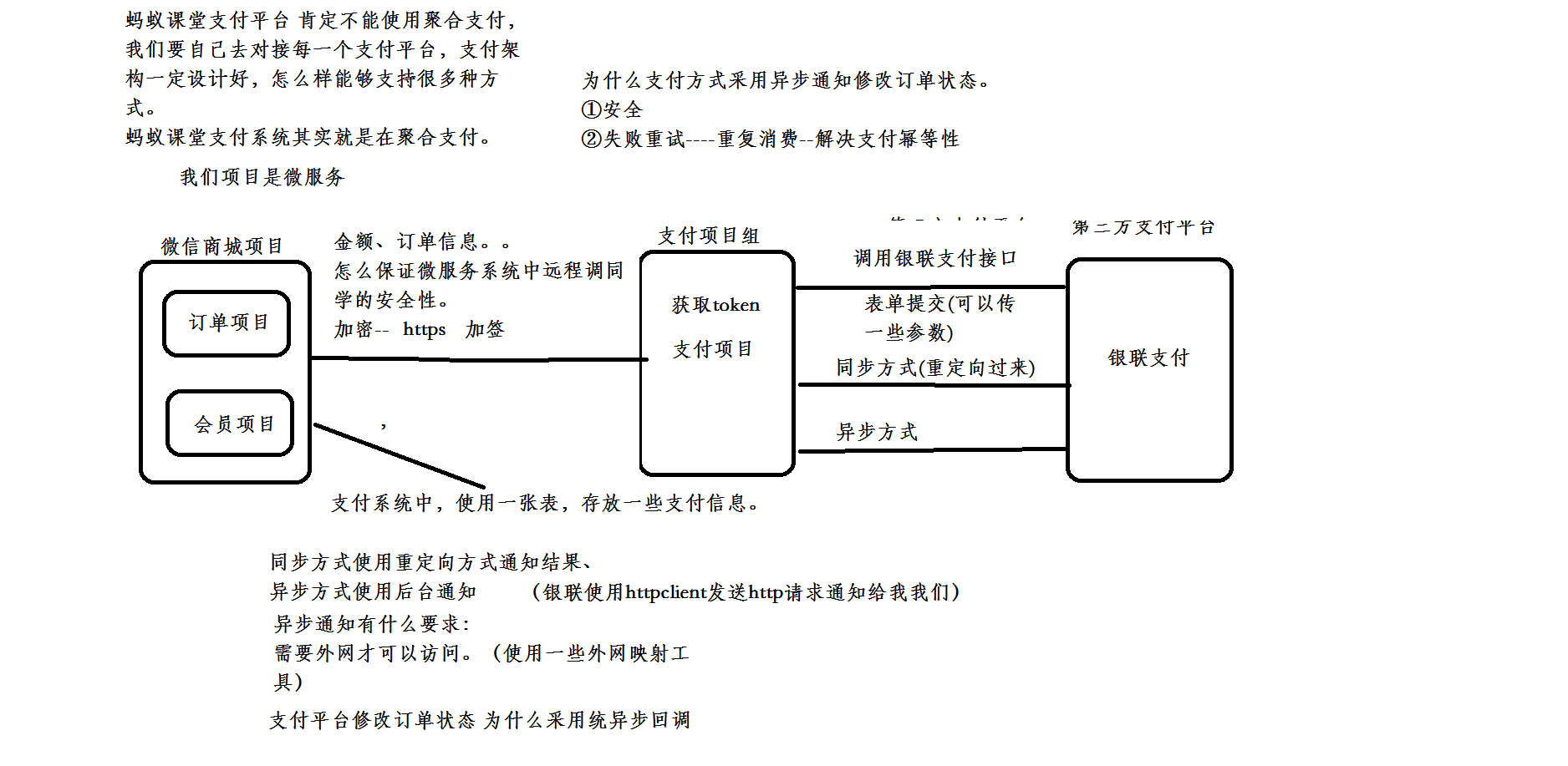
　　通过该例子可以发现，2PC协议存在明显的问题。假如D一直不能回复邮件，那么A、B和C将不得不处于一直等待的状态。并且B和C所持有的资源，即下周三不能安排其它活动，一直不能释放。其它等待该资源释放的活动也将不得不处于等待状态。

　　基于此，后来有人提出了三阶段提交协议，在其中引入超时的机制，将阶段1分解为两个阶段：在超时发生以前，系统处于不确定阶段；在超市发生以后，系统则转入确定阶段。

　　2PC协议包含协调者和参与者，并且二者都有发生问题的可能性。假如协调者发生问题，我们可以选出另一个协调者来提交事务。例如，班长组织活动，如果班长生病了，我们可以请副班长来组织。如果协调者出问题，那么事务将不会取消。例如，班级活动希望每个人都能去，假如有一位同学不能去了，那么直接取消活动即可。或者，如果大多数人去的话那么活动如期举行（2PC变种）。为了能够更好地解决实际的问题，2PC协议存在很多的变种，例如：树形2PC协议 （或称递归2PC协议）、动态2阶段提交协议（D2PC）等。

# 项目问题

## 支付项目支付流程



## 支回调怎么保证幂等性

**产生：第三方支付网关，重试机制造成幂等性**

**判断支付结果标识 注意：回调接口中，如果调用耗时代码，使用mq异步推送**

## 支回调数据安全性

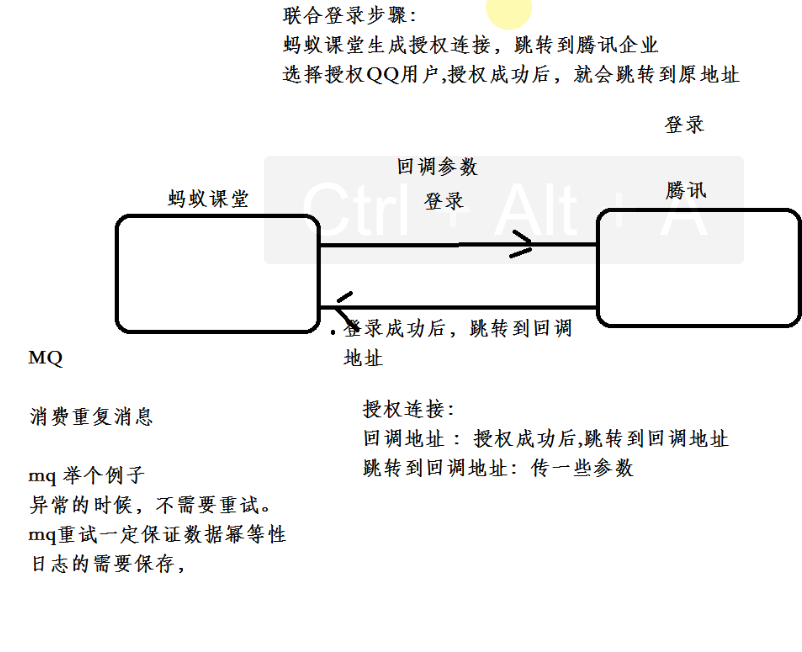
**Token、对称加密 base64 加签、rsa**

## 正回调中，项目宕机。

**使用对账（第三方支付交易接口），去进行查询。**

**对账（第三方交易平台交易结果）**

## 网页授权OAuth2.



**联合登录步骤：**

**蚂蚁课堂生成授权连接，跳转到腾讯企业**

**选择授权QQ用户,授权成功后，就会跳转到原地址**

**授权连接：**

**回调地址 ：授权成功后,跳转到回调地址**

**跳转到回调地址：传一些参数**

**跳转到回调地址：**

**传一个授权code有效期 10分钟 授权code使用完毕之后，直接删除，不能重复使用**

**授权码的作用：使用授权码换取aess\_token，使用aess\_token换取openid**

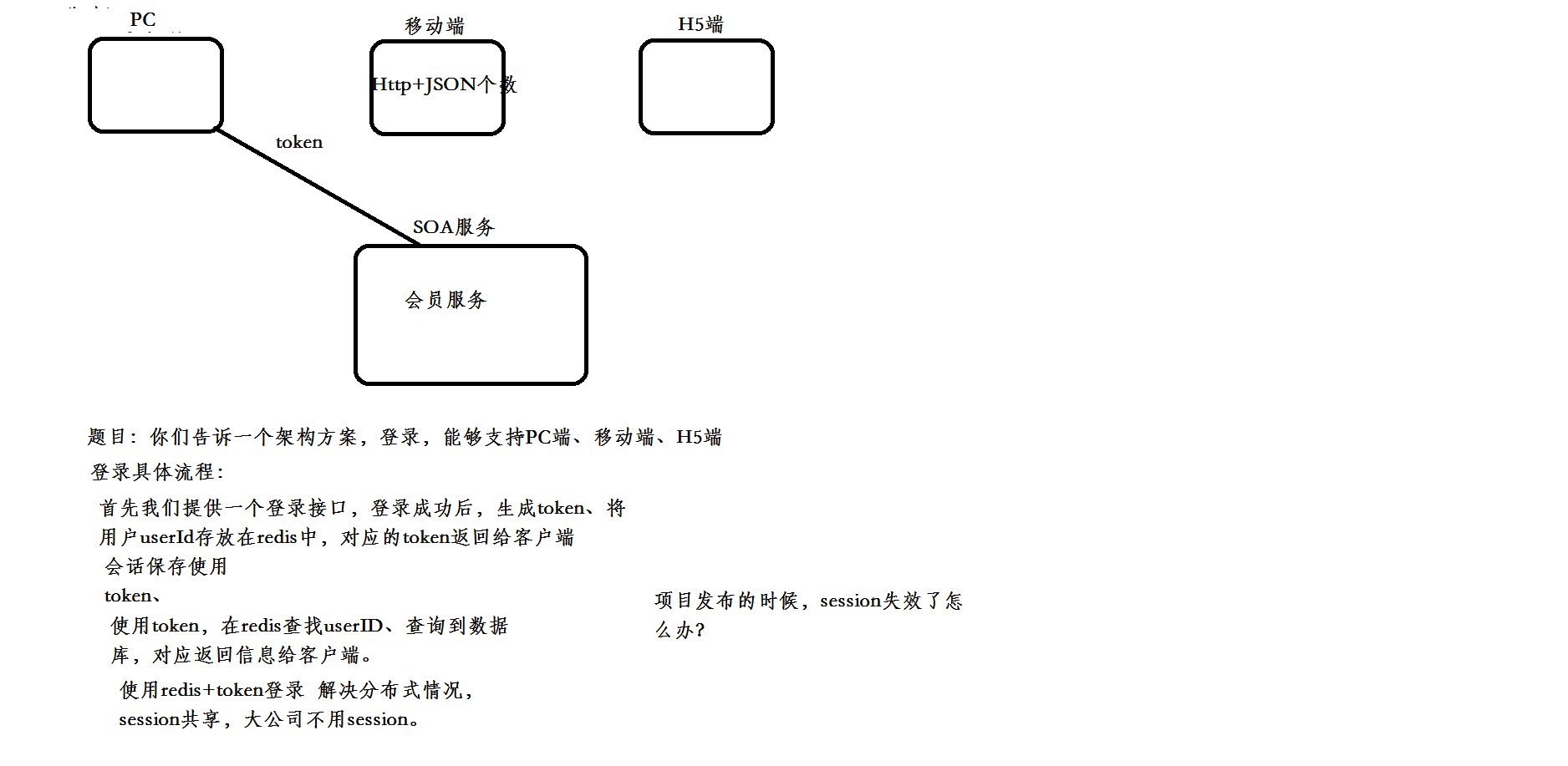
**openid作用: 唯一用户主键（授权系统会员主键，不代码腾讯userid）**

**openid和我们用户表中存放一个openid进行关联**

**使用openid调用腾讯会员接口查询QQ信息**

**本地回调**

## 你们登录流程



## 你在开发中遇到了那些难题，是怎么解决的？

### 跨域

**跨域原因产生：在当前域名请求网站中，默认不允许通过ajax请求发送其他域名。**

**XMLHttpRequest cannot load 跨域问题解决办法**

#### 使用后台response添加header

**后台response添加header，response.setHeader("Access-Control-Allow-Origin", "\*"); 支持所有网站**

#### 使用JSONP

**JSONP的优缺点:**

**JSONP只支持get请求不支持psot请求**

**什么是SQL语句注入**

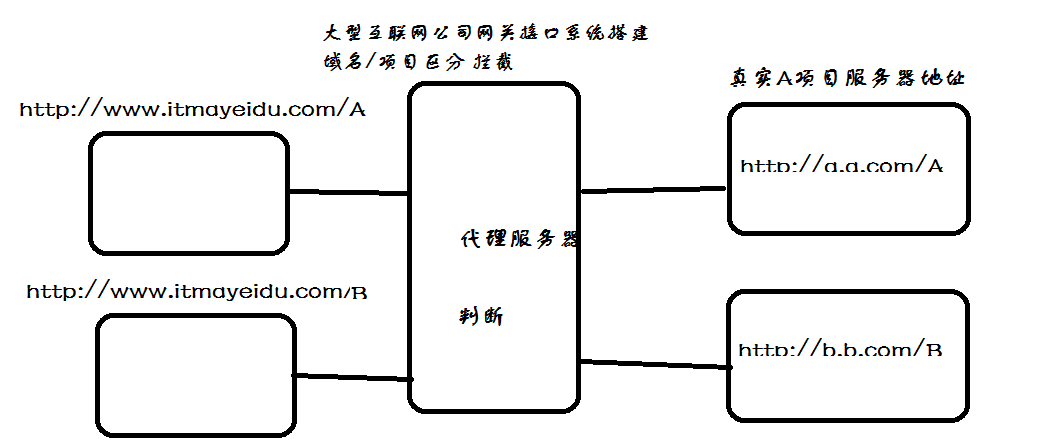
#### 使用接口网关

**使用nginx转发。**

**配置:**

|  |
| --- |
| server {  listen 80;  server\_name www.itmayiedu.com;  location /A {  proxy\_pass http://a.a.com:81/A;  index index.html index.htm;  }  location /B {  proxy\_pass http://b.b.com:81/B;  index index.html index.htm;  }  } |

**相关图:**



#### 使用内部服务器转发

**内部服务器使用HttpClient技术进行转发**

### 同步接口中保证数据一致性问题

例如A调用B接口，B接口没有及时反应，怎么进行补偿？

日志记录，任务调度定时补偿，自动重试机制。

### 任务调度幂等性问题

①使用zookeeper实现分布式锁 缺点(需要创建临时节点、和事件通知不易于扩展)

②使用配置文件做一个开关 缺点发布后，需要重启

③数据库唯一约束，缺点效率低

④使用分布式任务调度平台

XXLJOB

### MQ幂等性问题

解决办法:

1. 使用日志+msg-id保存报文信息，作为去重和幂等的依据。
2. 消费端代码抛出异常，不需要重试补偿，使用日志记录报文，下次发版本解决。