**PHP面试题目及答案要点**

1. 设计模式的六大原则
2. **开放封闭原则**

一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。 一个类应该是对扩展是开放的，对类的内部修改是封闭的，不应该通过修改来增加类的功能，而是通过扩展来增加功能。比如一个类中有两个功能，现在要增加第三个功能，那么这个新增加的功能不能对另外两个功能有影响。开放封闭原则是面向对象中最重要的原则，相当于一个总纲领。

1. **里氏替换原则**

所有引用基类的地方必须能透明地使用其子类的对象。继承必须确保父类中所拥有的所有性质在子类中依然成立。就是子类可以扩展父类的功能，子类尽量添加新的方法不要改变父类原有的功能，程序中对于实例化对象的子类型，不需要修改代码，可以直接进行替换。

有个著名的说法：正方形不是长方形。

 Liskov于1987年提出了一个关于继承的原则 “Inheritance should ensure that any property proved about supertype objects also holds for subtype objects.”（继承必须确保超类所拥有的性质在子类中仍然成立）。

 就是说，当一个子类的实例应该能够替换任何其父类的实例时，它们之间才具有is-A关系。

1. **依赖倒置原则**

高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。这条原则说明两个基本的要点：高阶的模块不应该依赖低阶的模块，它们都应该依赖于抽象；抽象不应该依赖于实现，实现应该依赖于抽象。一个类不应该强依赖另外一个类，每个类对于另外一个类都是可以替换的。高层模块不需要依赖底层的模块。比如A类和B类，A类需要B类的相关功能，A类不应该在其内部中直接调用B类，而是应该使用依赖注入的方式通过注入的方式，来将B类的对象注入到A类中，这样B类对于A类来说是可以替换的。高层模块也不应该依赖低层模块，二者都应该依赖其抽象，都是面向接口编程，而不是面向具体实现。接口一般都是稳定的。使用接口或者抽象类可以更好的制定一些规范和契约，而不去涉及具体的操作，具体的实现交给具体的类来完成。

**4. 单一职责原则** 不要存在多于一个导致类变更的原因。通俗的说，即一个类只负责一项职责。特点：一个类只负责一件事，可以降低类的复杂性。提高类的可读性，可维护性就提高了。降低变更引起可能出现的风险，如果接口的单一职责做得好，一个接口修改只对应的实现类有影响，对其他的接口无影响，这对系统的扩展性、维护性都有非常大的帮助。

    例如我们 web开发中常用的 MVC 开发模式就是单一职责原则的体现。Model 层提供数据。View 层专注于前端视图的展示。Controller层专注业务逻辑。

1. **接口隔离原则**

客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。 为各个类建立他们所需要的专门的接口。把一个接口切分为多个接口，把一个大的职责切分为小职责以及这些职责之间的协作交互。比如一个客户端不应该依赖它不需要的接口，即一个类对另外一个类的依赖应该建立在最小接口上。

    1).接口要尽量小。

    这是接口隔离原则的核心定义，不出现臃肿的接口（Fat Interface），但是“小”是有限度的，首先就是不能违反单一职责原则。

根据接口隔离原则拆分接口时，首先必须满足单一职责原则。

    2).接口要高内聚。

    高内聚就是要提高接口、类、模块的处理能力，减少对外的交互。具体到接口隔离原则就是，要求在接口中尽量少公布public方法，接口是对外的承诺，承诺地越少对系统开发越有利，变更的风险也就越少，同时也有利于降低成本。

    3).定制服务。

　定制服务就是单独为一个个体提供优良的服务。

    4).接口设计是有限度的。

    接口的设计粒度越小，系统越灵活，这是不争的事实。但是，灵活的同时也带来了结构的复杂化，开发难度增加，可维护性降低，这不是一个项目或产品所期望看到的，所以接口设计一定要注意适度，这个度只能根据经验和常识判断，没有一个固化或可测量的标准。

1. **迪米特法则（最少知识原则）**

一个对象应该对其他对象保持最少的了解。定义：只与你的直接朋友交谈，不跟“陌生人”说话。其含义是：如果两个类之间无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，尽量减少对其他类的依赖，提高模块的相对独立性。通俗的来讲，就是描述一个类对自己依赖的类知道的越少越好，无需关心是怎么实现的，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。

迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：迪米特法则中的“朋友”是指：当前对象本身、当前对象的成员对象、当前对象所创建的对象、当前对象的方法参数等，这些对象同当前对象存在关联、聚合或组合关系，可以直接访问这些对象的方法。只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。 所以一般应用迪米特法则设计的类都有一些特点，减少暴露类成员，类本身不能被改变，降低对类的访问权限等。

设计模式的门面模式（Facade）和中介者模式（Mediator），都是迪米特法则应用的例子。

优点：降低了类之间的耦合度，提高了模块的相对独立性。由于亲合度降低，从而提高了类的可复用性和系统的扩展性。

缺点：过分的使用迪米特原则，会产生大量这样的中间类和传递类，导致系统复杂度变大。

使用原则：使用迪米特原则时候要权衡好，保证系统架构的清晰。

1. **合成复用原则**

尽量使用组合或者聚合等关联关系来实现，其次在考虑使用继承关系来实现。

    组合是Has-A的关系，继承是Is-A关系。“Is-A”是严格的分类学意义上的定义，意思是一个类是另以个类的“一种”。而“Has-A”表示某一个角色具有某一项责任。比如狗和动物是继承关系，狗不能和动物用组合，因为狗不是动物组成的。

   合成复用原则和里氏替换原则是相辅相成的，如果要使用继承关系，则必须严格遵循里氏替换原则。

    通常类的复用分为继承复用和合成复用两种，继承复用虽然有简单和易实现的优点，但它也存在以下缺点：

· 继承复用破坏了类的封装性。因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。

· 子类与父类的耦合度高。父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护。

· 它限制了复用的灵活性。从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化。

   采用组合或聚合复用时，可以将已有对象纳入新对象中，使之成为新对象的一部分，新对象可以调用已有对象的功能，它有以下优点：

· 它维持了类的封装性。因为成分对象的内部细节是新对象看不见的，所以这种复用又称为“黑箱”复用。

· 新旧类之间的耦合度低。这种复用所需的依赖较少，新对象存取成分对象的唯一方法是通过成分对象的接口。

· 复用的灵活性高。这种复用可以在运行时动态进行，新对象可以动态地引用与成分对象类型相同的对象。

1. **其他原则**

· 配置化：尽可能的使用配置，而不要使用硬编码的方式。比如像一些数据库参数及常量，应该放到配置文件中。

· 面向接口编程：只需要关心接口，不需要关心具体的实现。

· DRY(Don't Repeat Yourself )避免重复代码：抽取重复的代码封装成一个函数或者类方法，保持里面的逻辑一致，可以被其他使用者调用。

1. PHP常见的设计模式
2. 工厂设计模式

要是当操作类的参数变化时，只用改相应的工厂类就可以工厂设计模式常用于根据输入参数的不同或者应用程序配置的不同来创建一种专门用来实例化并返回其对应的类的实例。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

**使用场景**：使用方法 new实例化类，每次实例化只需调用工厂类中的方法实例化即可。

**优点**：由于一个类可能会在很多地方被实例化。当类名或参数发生变化时，工厂模式可简单快捷的在工厂类下的方法中 一次性修改，避免了一个个的去修改实例化的对象。

例1：简单工厂

<?php

/\*简单工厂模式\*/

interface DB{

public function coon();

}

class mysqlDb implements DB {

public function conn(){

echo "连接到MySQL数据库";

}

}

class mongodb implements DB{

public function conn(){

echo "连接到mongodb数据库";

}

}

class DB\_factory{

static $db\_obj = null;

public static function createDB($type){

switch($type){

case 'mysql':

self::$db\_obj = new mysqlDb();

break;

case 'mongodb':

self::$db\_obj = new mongodb();

break;

default:

exit("unknow $type database");

}

return sefl::$db\_obj;

}

}

// 链接MySQL

$mysqlClient = DB\_factory::createDB('mysql');

$mysqlClient->conn();

//链接mongodb

$mongodbClient = DB\_factory::createDB('mongodb');

$mongodbClient->conn();

例2：抽象工厂

<?php

/\*抽象工厂模式\*/

interface DB{

public function conn();

}

interface Factory{

public function createDB();

}

/\*实现DB的MySQL类\*/

class mysqldb implements DB{

public funciton conn(){

echo "连接到MySQL数据库";

}

}

/\*实现DB的mongodb类\*/

class mongodb implements DB{

public function conn(){

echo "连接到mongodb数据库";

}

}

/\*创建对应的数据库工厂类\*/

class mysqlFactory implements Factory{

public function createDB(){

return new mysqldb();

}

}

class mongodbFactory implements Factory{

public function createDB(){

return new mongodb();

}

}

////////////////////

// 假设新增sqlite数据库 //

////////////////////

class sqliteDB implements DB{

public function conn(){

echo "连接到sqlite数据库";

}

}

/\*新增对用的工厂方法\*/

class sqlietFactory implements Factory{

public function createDB{

return new sqliteDB();

}

}

//测试

$mysqlfactory = new mysqlFactory();

$mysqldb = $mysqlfactory->createDB();

$mysqldb->conn();

$sqlitefactory=new sqliteFactory();

$sqlitedb=$sqlitefactory->createDB();

$sqlitedb->conn();

$mongofactory=new mongoFactory();

$mongodb=$mongofactory->createDB();

$mongodb->conn();

1. 单例模式

所谓单例模式，即在应用程序中最多只有该类的一个实例存在，一旦创建，就会一直存在于内存中！

**应用场景**：单例设计模式常应用于数据库类设计，采用单例模式，只连接一次数据库，防止打开多个数据库连接。

一个单例类应**具备以下特点**：单例类不能直接实例化创建，而是只能由类本身实例化。因此，要获得这样的限制效果，构造函数必须标记为private，从而防止类被实例化。

需要一个私有静态成员变量来保存类实例和公开一个能访问到实例的公开静态方法。

在PHP中，为了防止他人对单例类实例克隆，通常还为其提供一个空的私有\_\_clone()方法。

单例模式可以保证系统中一个类只有一个实例。即一个类只有一个对象实例。

/\*单例设计模式\*/

/\*

三私一公:

私有的保存唯一对象的属性;

私有的构造方法,防止类外new对象;

私有的克隆方法,防止外部clone对象;

公共的创建实例的方法.

\*/

class single{

// 保存唯一对象

private static $selfObj = null;

// 封锁外部new操作

private function \_\_construct(){};

// 封锁外部clone

private function \_\_clone(){};

public static function getObj(){

if(self::$selfObj === null){

self::$selfObj = new self();

}

return self::$getObj;

}

}

$q = single::getObj();

$w = single::getObj();

// true

var\_dump($q == $w);

1. 观察者模式

一个对象通过提供方法允许另一个对象即观察者 注册自己使本身变得可观察。当可观察的对象更改时，它会将消息发送到已注册的观察者。这些观察者使用该信息执行的操作与可观察的对象无关。结果是对象可以相互对话，而不必了解原因。观察者模式是一种事件系统，意味着这一模式允许某个类观察另一个类的状态，当被观察的类状态发生改变的时候，观察类可以收到通知并且做出相应的动作;观察者模式为您提供了避免组件之间紧密耦。

观察者设计模式定义了对象间的一种一对多的依赖关系，以便一个对象的状态发生变化时，所有依赖于它的对象都得到通知并自动刷新。

**观察者**：（Observer）将自己注册到被观察对象（Subject）中，被观察对象将观察者存放在一个容器（Container）里。

**被观察者**：被观察对象发生了某种变化，从容器中得到所有注册过的观察者，将变化通知观察者。

**撤销观察**：观察者告诉被观察者要撤销观察，被观察者从容器中将观察者去除。

**使用场景**：用户登录，需要写日志，送积分，参与活动等使用消息队列，把用户和日志、积分、活动之间解耦合。

**示例**：以下实例展示了安全验证和广告推送这两个观察者观察用户登录，登录后用户通知各个观察者执行自己的代码。

// PHP5.1以后提供SplObserver与被观察者SplSubject的接口

// 定义被观察者,实现SplSubject被观察者对象

class User implements SplSubject{

// 登陆次数

public $loginNum;

// 用户爱好

public $hobby;

// 用户姓名

public $name;

public function \_\_construct($name,$hobby = ''){

$this->loginNum = rand(1,10);

$this->hobby = $hobby;

$this->name = $name;

// 用来保存观察者对象

$this->observers = new SplObjectStorage();

}

// 用户登录入口

public function login(){

echo "用户登录成功后通知: ";

$this->notify();

}

// 添加观察者

public function attach(SplObserver $observer){

$this->observers->attach($observer);

}

// 删除观察者

public function detach(SplObserver $observer){

$this->observers->detach($observer);

}

//遍历观察者发送通知

public function notify(){

//指针指向头部

$this->observers->rewind();

while ($this->observers->valid()) {

// 当前观察者

$observer=$this->observers->current();

// var\_dump($observer);

// 传递当前用户

$observer->update($this);

$this->observers->next();

}

}

}

// 定义安全验证观察者

class Secrity implements SplObserver{

/\*\*

\* 本观察者通知的具体方法

\* @param SplSubject $subject [被观察者user]

\*/

public function update(SplSubject $subject){

if($subject->loginNum <= 3){

echo $subject->name.',这是第'.$subject->loginNum.'登录';

}else{

echo $subject->name.',这是第'.$subject->loginNum.'登录，登录次数过多.';

}

}

}

//定义广告推荐观察者

class Advertise implements SplObserver{

public function update(SplSubject $subject){

if($subject->hobby == 'car'){

echo "给你推荐汽车广告 ".PHP\_EOL;

}else if($subject->hobby=='php'){

echo '推荐php培训广告'.PHP\_EOL;

}else{

echo '好好学习，天天向上'.PHP\_EOL;

}

}

}

// 测试

$ming=new User('小明','php');

// 添加观察者

$secrity=new Secrity();

$ming->attach($secrity);

$commend=new Advertise();

$ming->attach($commend);

$ming->login();

echo '-------------------------------'.PHP\_EOL;

$hong=new User('小红');

// 添加观察者

$secrity=new Secrity();

$hong->attach($secrity);

$commend=new Advertise();

$hong->attach($commend);

// 删除广告推荐

$hong->detach($commend);

$hong->login();

1. 适配器模式

将一个类的接口转换成客户希望的另一个接口,适配器模式使得原本的由于接口不兼容而不能一起工作的那些类可以一起工作。  
 **应用场景**：老代码接口不适应新的接口需求，或者代码很多很乱不便于继续修改，或者使用第三方类库。

**示例**：php连接数据库的方法：mysql,,mysqli,pdo,可以用适配器一。

// 原来代码

class User{

private $name;

function \_\_construct($name){

$this->name = $name;

}

public function get\_name(){

echo $this->name;

}

}

// 新代码,开放平台标准接口

interface UserInterface{

function getUserName();

}

class UserInfo implements UserInterface{

protected $user;

function \_\_construct($user){

$this->user = $user;

}

public function getUserName(){

return $this->user->get\_name();

}

}

$old\_user = new User('张三');

echo $old\_user->get\_name()."\n";

$new\_user = new UserInfo($old\_user);

echo $new\_user->getUserName()."\n";

1. 策略模式

将一组特定的行为和算法封装成类，以适应某些特定的上下文环境。

**使用场景**：策略模式是依赖注入，控制反转的基础

**示例**：一个电商网站系统，针对男性女性用户要各自跳转到不同的商品类目，并且所有广告位展示不同的广告

MaleUserStrategy.php

<?php

class MaleUserStrategy implements UserStrategy {

function showAd()

{

echo "IPhone6";

}

function showCategory()

{

echo "电子产品";

}

}

FemaleUserStrategy.php

<?php

namespace IMooc;

class FemaleUserStrategy implements UserStrategy {

function showAd()

{

echo "2014新款女装";

}

function showCategory()

{

echo "女装";

}

}

UserStrategy.php

<?php

namespace IMooc;

interface UserStrategy {

function showAd();

function showCategory();

}

<?php

interface FlyBehavior{

public function fly();

}

class FlyWithWings implements FlyBehavior{

public function fly(){

echo "Fly With Wings \n";

}

}

class FlyWithNo implements FlyBehavior{

public function fly(){

echo "Fly With No Wings \n";

}

}

class Duck{

private $\_flyBehavior;

public function performFly(){

$this->\_flyBehavior->fly();

}

public function setFlyBehavior(FlyBehavior $behavior){

$this->\_flyBehavior = $behavior;

}

}

class RubberDuck extends Duck{

}

// Test Case

$duck = new RubberDuck();

/\* 想让鸭子用翅膀飞行 \*/

$duck->setFlyBehavior(new FlyWithWings());

$duck->performFly();

/\* 想让鸭子不用翅膀飞行 \*/

$duck->setFlyBehavior(new FlyWithNo());

$duck->performFly();

1. 装饰器模式

**使用场景**：当某一功能或方法draw，要满足不同的功能需求时，可以使用装饰器模式；

**实现方式**：在方法的类中建addDecorator(添加装饰器),beforeDraw,afterDraw 3个新方法， 后2个分别放置在要修改的方法draw首尾。然后创建不同的装器类（其中要包含相同的,beforeDraw,afterDraw方法）能过addDecorator添加进去，然后在beforeDraw,afterDraw中循环处理，与观察者模式使用有点相似：  
 1.装饰器模式（Decorator），可以动态地添加修改类的功能  
 2.一个类提供了一项功能，如果要在修改并添加额外的功能，传统的编程模式，需要写一个子类继承它，并重新实现类的方法  
 3.使用装饰器模式，仅需在运行时添加一个装饰器对象即可实现，可以实现最大的灵活性

DrawDecorator.php

<?php

namespace IMooc;

interface DrawDecorator

{

function beforeDraw();

function afterDraw();

}

Canvas.php

<?php

namespace IMooc;

class Canvas

{

public $data;

protected $decorators = array();

//Decorator

function init($width = 20, $height = 10)

{

$data = array();

for($i = 0; $i < $height; $i++)

{

for($j = 0; $j < $width; $j++)

{

$data[$i][$j] = '\*';

}

}

$this->data = $data;

}

function addDecorator(DrawDecorator $decorator)

{

$this->decorators[] = $decorator;

}

function beforeDraw()

{

foreach($this->decorators as $decorator)

{

$decorator->beforeDraw();

}

}

function afterDraw()

{

$decorators = array\_reverse($this->decorators);

foreach($decorators as $decorator)

{

$decorator->afterDraw();

}

}

function draw()

{

$this->beforeDraw();

foreach($this->data as $line)

{

foreach($line as $char)

{

echo $char;

}

echo "<br />\n";

}

$this->afterDraw();

}

function rect($a1, $a2, $b1, $b2)

{

foreach($this->data as $k1 => $line)

{

if ($k1 < $a1 or $k1 > $a2) continue;

foreach($line as $k2 => $char)

{

if ($k2 < $b1 or $k2 > $b2) continue;

$this->data[$k1][$k2] = ' ';

}

}

}

}

Index.php

<?php

define('BASEDIR', \_\_DIR\_\_);

include BASEDIR.'/IMooc/Loader.php';

spl\_autoload\_register('\\IMooc\\Loader::autoload');

$canvas = new IMooc\Canvas();

$canvas->init();

$canvas->addDecorator(new \IMooc\ColorDrawDecorator('green'));

$canvas->rect(3,6,4,12);

$canvas->draw();

1. 依赖注入(DI)与控制反转(IoC)
2. 概念

**容器**：常见的变量、对象属性等都可以算是容器。一个容器能够装什么，全部取决于你对该容器的定义。当然，现在我们讨论的是这样一种容器，它存放的不是文本、数值，而是对象、对象的描述（类、接口）或者是提供对象的回调(闭包)，通过这种容器，我们得以实现许多高级的功能，其中最常提到的，就是 “解耦”、“依赖注入”。

**IOC - Inversion of Control 控制反转**：控制反转是从容器的角度在描述，即容器控制应用程序，由容器反向的向应用程序注入应用程序所需要的外部资源。

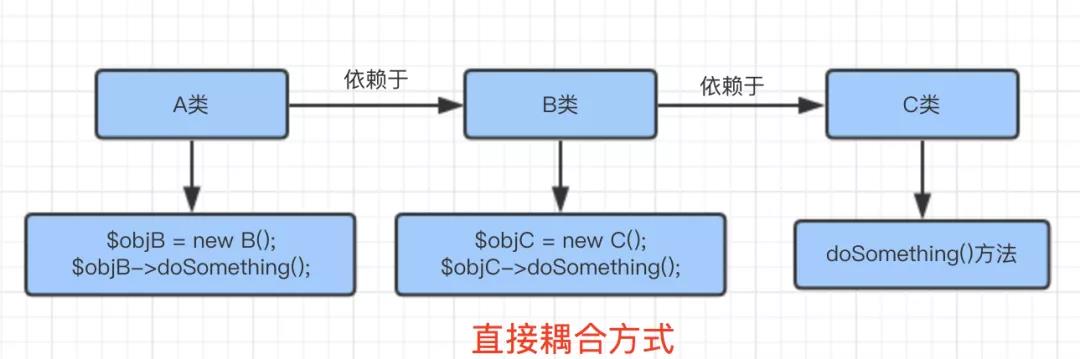
**DI - Dependency Injection 依赖注入**：依赖注入是从应用程序的角度在描述，可以把依赖注入理解为：应用程序依赖容器创建并注入它所需要的外部资源。

备注：依赖注入和控制反转说的是同一个东西，是一种设计模式，这种设计模式用来减少程序间的耦合，从某个方面讲，就是它们描述的角度不同。

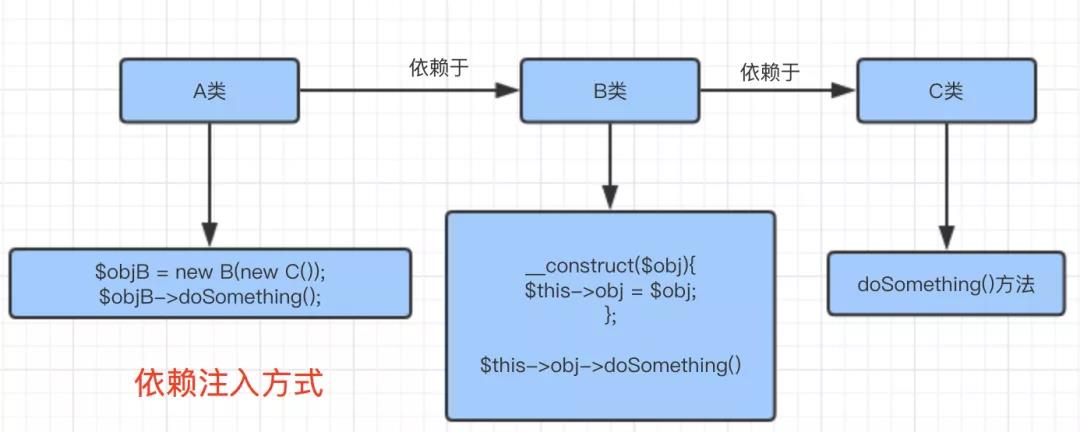
1. 原理

一般情况下，当存在类与类之间的依赖关系的时候，我们都是通过直接实例化的方式进行调用。一旦出现多层依赖，这种方式的耦合程度就很高，在需要修改其中一个类的时候，会牵扯很多依赖它的类的修改，因此对代码的改动会比较大。

下面简单举一个A->B->C三层依赖的关系解释怎么运用依赖注入来解耦，提高开发效率。



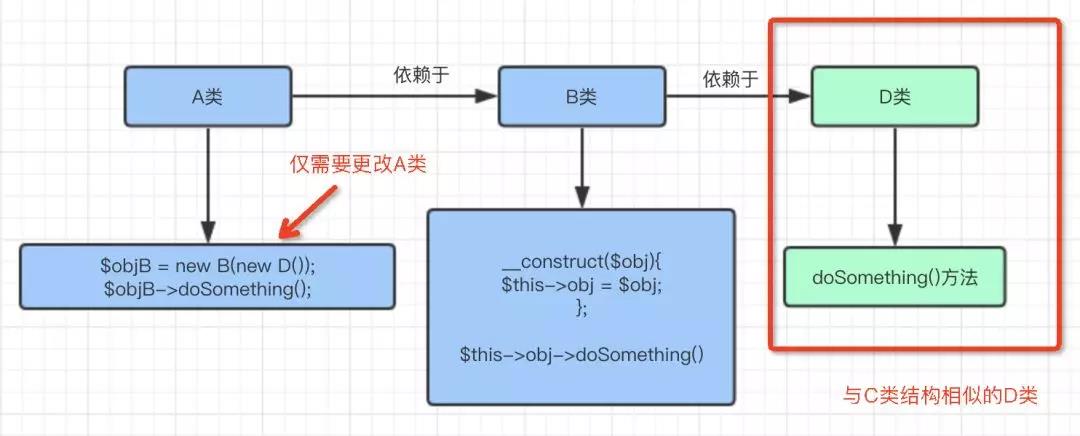
而依赖注入方式如下：



解析：

常规写法里面，一旦C类需要作出改变，或者B类的调用需要改变成D类的时候，还需要考虑到依赖自己的B类，即还需要对B类作出修改。

依赖注入的思想就是即用即实例，反转类与类之间的控制关系，实现由调用类A类控制后续的依赖关系，这样可以让B类随意的更改所需依赖和实例化的类（C类或D类），达到解耦的目的。



1. 常用方式

1、构造方法注入；2、set属性注入；3、静态工厂方法注入；

上述的例子使用的就是构造方法注入的方式，将对象作为参数传递到构造方法中；同样的set属性注入也是相类似的方法，不同的仅仅是在set一个类的成员的属性时传递这个对象参数，在此就不一一举例了。

除此之外，还有静态工厂方法注入的方式，这种方法与静态工厂方法类似。静态工厂方法就是通过一个类来管理需要实例化的多个相似的类，该类会定义一个方法用于获取需要实例化的对象，而具体要实例化哪个对象就依赖于传递进来的对象名参数了。

对于静态工厂方式的注入，与一般的静态工厂方法不同之处在于这个传进来的参数是一个已经实例化过的对象。

<?php

class IoC{

protected static $registry = [];

public static function bind($name, Callable $resolver) //传入类名和类对象实例

{

static::$registry[$name] = $resolver;

}

public static function make($name) //静态工厂方法

{

if (isset(static::$registry[$name])) {

$resolver = static::$registry[$name];

return $resolver(); //实例化

}

throw new Exception('Alias does not exist in the IoC registry.');

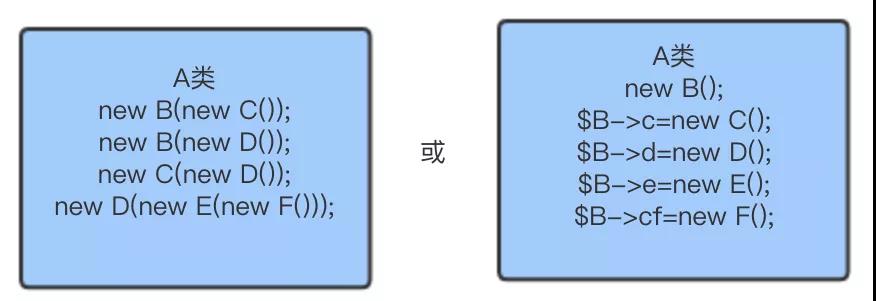
}

}

总而言之，三种方式传递的都是实例化对象，只是不同之处在于传递的位置分别为构造方法、set属性、静态工厂方法而已。

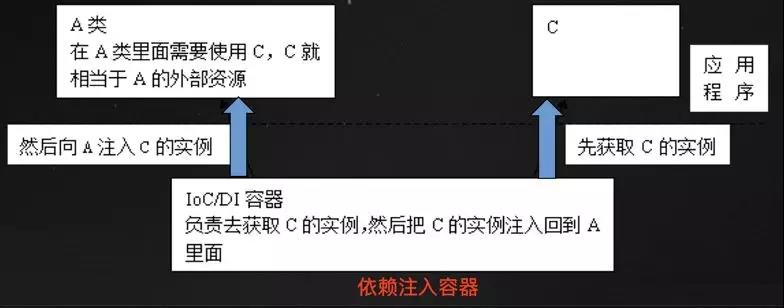
1. 依赖注入容器（IoC容器）

大多数时侯，在使用依赖注入方式解耦组件时，并不需要用到容器。  
当一段程序需要实例化的类太多或者依赖太多的时候，重复依赖注入的代码是比较繁琐的事情，例如以下情况：



当产生以上关系的时候，依赖注入的代码会比较混乱，而且存在重复，更有可能在调用一个一般方法时new一个不需要的类，产生冗余。

此时需要使用容器，使用依赖注入容器后的思路是应用程序需要到A类，就从容器内取得A类。具体是容器创建C类，再创建B类并把C注入，再创建A类，并把B类注入，应用程序调用A类方法， A类调用B类方法，接着做些其它工作.总之容器负责实例化，注入依赖，处理依赖关系等工作。



对于实际开发中复杂多变的代码环境，我们并不能完全知道现在的类在未来会扩展成什么情况，因此我们需要在有新的依赖类加入的时候，通过容器去实现实例化该类的方法。因此，在实例化未知类的时候，最能探索一个类的内部结构和实例化的方法就是利用**反射**，由此可知，**反射是容器管理各个依赖类的核心**。我们可以通过实例来了解容器的内部实现：

三个存在依赖关系的类：文件testClass.php

<?php

//依赖关系：Company->Department->Group

class Group{

public function doSomething(){

echo \_\_CLASS\_\_.":".'hello', '|';

}

}

class Department{

private $group;

public function \_\_construct(Group $group){

$this->group = $group;

}

public function doSomething(){

$this->group->doSomething();

echo \_\_CLASS\_\_.":".'hello', '|';

}

}

class Company{

private $department;

public function \_\_construct(Department $department){

$this->department = $department;

}

public function doSomething(){

$this->department->doSomething();

echo \_\_CLASS\_\_.":".'hello', '|';

}

}

Ioc容器的内部实现：

<?php

class Container

{

private $s = array();

public function \_\_set($k, $c)

{

$this->s[$k] = $c;

}

public function \_\_get($k)

{

return $this->build($this->s[$k]);

}

/\*\*

\* 自动绑定（Autowiring）自动解析（Automatic Resolution）

\*

\* @param string $className

\* @return object

\* @throws Exception

\*/

public function build($className)

{

// 如果是匿名函数（Anonymous functions），也叫闭包函数（closures）

if ($className instanceof Closure) {

// 执行闭包函数，并将结果

return $className($this);

}

/\*通过反射获取类的内部结构，实例化类\*/

$reflector = new ReflectionClass($className);

// 检查类是否可实例化, 排除抽象类abstract和对象接口interface

if (!$reflector->isInstantiable()) {

throw new Exception("Can't instantiate this.");

}

/\*\* @var ReflectionMethod $constructor 获取类的构造函数 \*/

$constructor = $reflector->getConstructor();

// 若无构造函数，直接实例化并返回

if (is\_null($constructor)) {

return new $className;

}

// 取构造函数参数,通过 ReflectionParameter 数组返回参数列表

$parameters = $constructor->getParameters();

// 递归解析构造函数的参数

$dependencies = $this->getDependencies($parameters);

// 创建一个类的新实例，给出的参数将传递到类的构造函数。

return $reflector->newInstanceArgs($dependencies);

}

/\*\*

\* @param array $parameters

\* @return array

\* @throws Exception

\*/

public function getDependencies($parameters)

{

$dependencies = [];

/\*\* @var ReflectionParameter $parameter \*/

foreach ($parameters as $parameter) {

/\*\* @var ReflectionClass $dependency \*/

$dependency = $parameter->getClass();

if (is\_null($dependency)) {

// 是变量,有默认值则设置默认值

$dependencies[] = $this->resolveNonClass($parameter);

} else {

// 是一个类，递归解析

$dependencies[] = $this->build($dependency->name);

}

}

return $dependencies;

}

/\*\*

\* @param ReflectionParameter $parameter

\* @return mixed

\* @throws Exception

\*/

public function resolveNonClass($parameter)

{

// 有默认值则返回默认值

if ($parameter->isDefaultValueAvailable()) {

return $parameter->getDefaultValue();

}

throw new Exception('I have no idea what to do here.');

}

}

require\_once "./testclass.php";

//开始测试，先测试已知依赖关系的情况

$c = new Container();

$c->department = 'Department';

$c->company = function ($c) {

return new Company($c->department);

};

// 从容器中取得company

$company = $c->company;

$company->doSomething();

//输出:

Group:hello|Department:hello|Company:hello|

// 测试未知依赖关系，直接使用的方法

$di = new Container();

$di->company = 'Company';

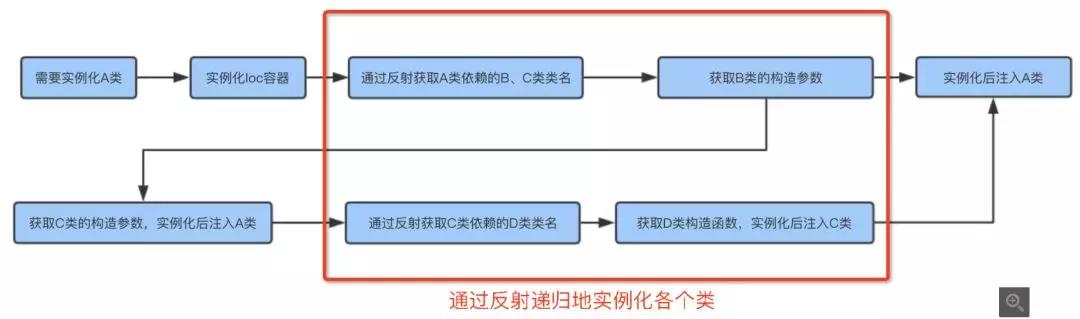
$company = $di->company;

$company->doSomething();

//输出:

Group:hello|Department:hello|Company:hello|

我们可以通过一张图解释Ioc容器的内部逻辑：



1. 总结

IOC的基本概念是：不创建对象，但是描述创建它们的方式。在代码中不直接与对象和服务连接，但在配置文件中描述哪一个组件需要哪一项服务。

1. ThinkPHP与laravel的区别

主要区别:(thinkPHP更适合国人的编码习惯)  
1. 渲染模版方式的不同:  
在Laravel框架里,使用return view()来渲染模版;而ThinkPHP里则使用了$this->display()的方式渲染模版;  
2. 在Laravel框架里,由于其考虑到了跨站请求伪造, 所以如果使用form表单以post方式进行传值时,如果不再form表单中加入{{csrf\_field()}}则会报出TokenMethodnotfound的语法错误;而TP框架则需要自己手动完成防止跨站攻击的代码;  
3. Laravel是一个重路由的框架(5.4),所有的功能都是由路由发起的,哪怕没有控制器方法,只要写了路由就能够访问,thinkPHP(3.2),必须要有控制器方法才能正常访问;  
4. laravel具有强大的社区化扩展,（composer扩展自动加载）;  
5. laravel具有强大的Blade模版引擎;  
6. 中间件，Laravel特点，可以实现访问前后的处理，例如请求和返回，权限认证等;  
7. 条件判断语句书写方式的差异:  
Laravel框架里 if else判断语句和foreach语句 书写时必须以@if开头 以@endif结尾,如果没有则报语法错误,@foreach @endforeach同理;  
而TP框架则和PHP语法规则使用方式一致直接ifesle语句判断和foreach循环遍历。

1. Cookie与Session

**Cookie通过在客户端记录信息确定用户身份，Session通过在服务器端记录信息确定用户身份。**

**Cookie机制**

由于HTTP是一种无状态的协议，服务器单从网络连接上无从知道客户身份。怎么办呢？就给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是Cookie的**工作原理**。

记录用户访问次数

Cookie的不可跨域名性

中文属于Unicode字符，在内存中占4个字符，而英文属于ASCII字符，内存中只占2个字节。Cookie中使用Unicode字符时需要对Unicode字符进行编码，否则会乱码。

Cookie不仅可以使用ASCII字符与Unicode字符，还可以使用二进制数据。

Cookie的maxAge决定着Cookie的有效期，单位为秒（Second）。Cookie中通过getMaxAge()方法与setMaxAge(int maxAge)方法来读写maxAge属性。

Cookie并不提供修改、删除操作。如果要修改某个Cookie，只需要新建一个同名的Cookie，添加到response中覆盖原来的Cookie。

domain属性决定运行访问Cookie的域名，而path属性决定允许访问Cookie的路径（ContextPath）。

**Session机制**

Cookie机制是通过检查客户身上的“通行证”来确定客户身份的话，那么Session机制就是通过检查服务器上的“客户明细表”来确认客户身份。Session相当于程序在服务器上建立的一份客户档案，客户来访的时候只需要查询客户档案表就可以了。

Session机制决定了当前客户只会获取到自己的Session，而不会获取到别人的Session。各客户的Session也彼此独立，互不可见。

Session保存在服务器端。为了获得更高的存取速度，服务器一般把Session放在内存里。每个用户都会有一个独立的Session。如果Session内容过于复杂，当大量客户访问服务器时可能会导致内存溢出。因此，Session里的信息应该尽量精简。

Session在用户第一次访问服务器的时候自动创建。

Session生成后，只要用户继续访问，服务器就会更新Session的最后访问时间，并维护该Session。

为防止内存溢出，服务器会把长时间内没有活跃的Session从内存删除。这个时间就是Session的超时时间。如果超过了超时时间没访问过服务器，Session就自动失效了。

**区别：**

1. cookie数据存放在客户的浏览器上，session数据放在服务器上；
2. cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗考虑到安全应当使用session。
3. 设置cookie时间可以使cookie过期。但是使用session-destory（），我们将会销毁会话。
4. session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能考虑到减轻服务器性能方面，应当使用cookie。
5. 单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。(Session对象没有对存储的数据量的限制，其中可以保存更为复杂的数据类型)

两者最大的区别在于生存周期，一个是IE启动到IE关闭.(浏览器页面一关 ,session就消失了)，一个是预先设置的生存周期，或永久的保存于本地的文件。(cookie)

1. Get与Post请求的区别

表单提交中get和post方式的区别：

1. get是从服务器上获取数据，post是向服务器传送数据。

2. get是把参数数据队列加到提交表单的ACTION属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。post是通过HTTPpost机制，将表单内各个字段与其内容放置在HTML HEADER内一起传送到ACTION属性所指的URL地址。用户看不到这个过程。

3.对于get方式，服务器端用Request.QueryString获取变量的值，对于post方式，服务器端用Request.Form获取提交的数据。

4.get传送的数据量较小，不能大于2KB。post传送的数据量较大，一般被默认为不受限制。但理论上，IIS4中最大量为80KB，IIS5中为100KB。（这里有看到其他文章介绍get和post的传送数据大小跟各个浏览器、操作系统以及服务器的限制有关）

5.get安全性非常低，post安全性较高。最直观的区别就是GET把参数包含在URL中，POST通过request body传递参数。

1. 常见的数据结构

数据结构就是指数据元素之间的关系，是复杂数据的组织方式。

数据结构三要素：逻辑结构（集合、线性、树形、网状（图））、存储结构（顺序存储、链式存储）、数据操作（算法，增删改查操作）。

Data\_Structure = (D，R) D：是数据元素的有限集，R：是D上关系有限集

1. 数组

**数组(Array)**是一种线性表的数据结构，它用一段连续的内存空间，来存储具有相同类型的值。但是由于在**PHP的底层定义中，数组是通过散列表实现的**，所以这段定义并不适用。PHP的数组可以存储任意数据类型的数据，所以相对于Java来说效率较高。在Java的数组中，每次定义都要先声明属于组的类型，在查找数组时，效率是O(1)，但是在插入和删除时，算法复杂度是O(n),因为在插入操作时，要先找到插入的位置，然后将该位置及往后的元素都往后移一位。删除同理。但是PHP却不受此约束。

1. 链表

和数组不同，链表并不需要一块连续的内存空间，它通过“指针”将一组零散的内存块串联起来使用，一般节点有两个属性（data和next）。链表有多种类型，最简单的是单链表。

**单链表**是最原始的链表。单链表有两个节点比较特殊，头结点和尾节点。头结点记录链表的基地址，通过它可以遍历得到整条链表。尾节点的指针不是指向节点，而是指向空地址NULL，表示这是最后一个节点。单向链表插入和删除的时间复杂度是O(1),而查询的时间复杂度是O(n)。  
疑问：当进行插入和删除操作时要先查询相应节点，查询的时间复杂度是O(n),为什么插入和删除的的复杂度是O(1)呢？可以将插入删除看作是单纯的插入删除，不包含查询在里面。当做两个不同的操作来看待。

在单链表的基础上扩展有了循环链表，循环链表是将尾节点的next指向了头结点，从而实现了收尾相连。可以解决(约瑟夫环)问题。

**双向链表**：与单向链表的区别是除了有一个指向下一个节点的指针外，还有一个用于指向上一个节点的指针。从而实现通过O(1)复杂度找到上一个节点。使得双向链表在插入删除是比单向链表更高效。以删除为例，在删除节点时，我们还要获取其前驱节点，让前驱节点的指针指向被删除节点的下一个节点。在单向链表中，获取前驱节点的复杂度是O(n)，但是双向链表O(1)直接获取前驱节点。所以双向链表插入和删除的时间复杂度才是真正的O(1)。

最后一种就是**双向循环链表**，就是双向链表和单向链表的结合。

时间复杂度是靠更差的空间复杂度换取的，双向链表始终需要单链表的两倍空间在 Web 应用中，时间效率优先级更高，所以我们通常都是空间换时间来提高性能。

1. 队列

和栈类似，队列也是一种特殊的线性表结构，只不过队列是在一端插入，另一端删除，就跟我们平常排队一样，从队尾入队，在队头出去，所以队列的特性是**先入先出**，允许插入的一端叫队尾，允许删除的一端叫队头。队列也可以通过数组和链表实现，通过数组实现的叫顺序队列，通过链表实现的叫做链式队列，栈只需要一个栈顶指针就可以了，因为只允许在栈顶插入删除，但是队列需要两个指针，一个指向队头，一个指向队尾。

PHP中array\_shift()和array\_push()来实现。

1. 栈

限定只能在一端进行插入和删除操作的线性表，并且满足**先进后出**的特点。我们把允许插入和删除的一端叫做栈顶，另一个端叫做栈底，不含任何数据的栈叫做空栈。栈支持通过数组/链表实现，通过数组实现的通常叫做顺序栈，通过链表实现的叫做链栈。

PHP中可通过array\_push()和array\_pop()来实现。

1. 常见算法及时间复杂度
2. 冒泡排序 O(n\*n)

冒泡排序（Bubble Sort）是一种简单的排序算法。它重复地走访过要排序的数列，依次比较两个元素，如果他们的顺序错误就把他们交换过来 。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

**步骤**：

（1）比较相邻的元素。如果第一个比第二个大，就交换他们两个。

（2）对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

（3）针对所有的元素重复以上的步骤，除了最后一个。

（4）持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

function bubbleSort($arr){

$len=count($arr);

//该层循环控制 需要冒泡的轮数

for($i=1;$i<$len;$i++){

//该层循环用来控制每轮 冒出一个数 需要比较的次数

for($j=0;$j<$len-$i;$j++){ //把小于号换成大于号就是，小数在前，大数在后

if($arr[$j]<$arr[$j+1]){

list($arr[$j+1],$arr[$j])=[$arr[$j],$arr[$j+1]];

}

}

}

return $arr;

}

// 另一种写法

function bubble\_sort($a){

$c = count($a);

for($i=0;$i<$c;$i++){

for ($j=$c-1;$j>$i;$j--){

if($a[$j] < $a[$j-1]){

$t = $a[$j-1];

$a[$j-1] = $a[$j];

$a[$j] = $t;

}

}

}

return $a;

}

1. 选择排序 O(n\*n)

选择排序在冒泡排序的基础上进行了改进，每次通过列表时只进行一次传递交换。简单来说，选择排序的原理就是 每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。 选择排序是不稳定的排序方法。

**步骤:**

（1）先假设最小值的位置。

（2）把当前假设的值与剩下的元素做比较。

（3）比较，发现更小的，记录下最小的位置；并在下次比较的时候，采用已知的最小值最比较。

（4）如果发现，最小值的位置与当前假设的最小值的位置不同，则位置互换。反之，假设成立，当前位置保留继续往下找，直到排序完成。

//实现思路 双重循环完成，外层控制轮数，当前的最小值。内层 控制的比较次数

function selectSort($arr){

$len = count($arr);

for($i = 0;$i < $len-1;$i++){

//先假设最小值的位置

$q=$i;

//当前需要和哪些元素比较 （$i 后面的）

for($j = $i+1;$j < $len;$j++){

// $arr[$q] 当前已知的最小值

if($arr[$q] > $arr[$j]){

//比较，发现更小的，记录下最小的位置；并在下次比较的时候，采用已知的最小值最比较

$q = $j;

}

}

//如果发现，最小值的位置与当前假设的位置的$i不同，则位置互换

if($q != $i){

list($arr[$i],$arr[$q])=[$arr[$q],$arr[$i]];

}

}

return $arr;

}

1. 插入排序 O(n\*n)

插入排序是一种逻辑上非常好理解的排序方式，整个排序的核心就是不断在当前已经排好部分数据的数组里，找到合适的位置插入新数据。就像抓扑克牌，抓一张，然后再手里已经部分已经排好序的手牌的找到位置插进去。

**步骤**：

（1）从第一个元素开始，该元素可以认为已经被排序；

（2）取出下一个元素，在已经排序的元素序列中从后向前扫描；

（3）如果该元素（已排序）大于新元素，将该元素移到下一位置；

（4）重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

（5）将新元素插入到该位置中；

（6）重复步骤2。

function insertSort($arr){

$len=count($arr);

//第一个元素可认定为已被排序

for($i=1;$i<$len;$i++){

//获取当前需要比较的元素值

$tmp=$arr[$i];

for ($j=$i-1; $j >=0 ; $j--) {

//$tmp 需要插入的元素 $arr[$j]需要比较的元素

if($arr[$j]>$tmp){

//发现当前插入元素要小，交换位置

list($arr[$j],$arr[$j+1])=[$arr[$j+1],$arr[$j]];

}

}

}

return $arr;

}

1. 快速排序 O(n\*log2(n))

快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序 n 个项目要Ο(n log n)次比较。在最坏状况下则需要Ο(n2)次比较，但这种状况并不常见。事实上，快速排序通常明显比其他Ο(n log n) 算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来，且在大部分真实世界的数据，可以决定设计的选择，减少所需时间的二次方项之可能性。

通过设置一个初始中间值，来将需要排序的数组分成3部分，小于中间值的左边，中间值，大于中间值的右边，继续递归用相同的方式来排序左边和右边，最后合并数组。

**步骤**:

1. 从数列中挑出一个元素，称为 “基准”（pivot），
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

function quick\_sort($arr){

//判断参数是否是一个数组  
if(!is\_array($arr)) return false;

$len = count($arr);

if($len <= 1){

return $arr;

}

$left\_arr = [];

$right\_arr = [];

for($i = 1;$i < $len;$i++){

if($arr[$i] < $arr[0]){

$left\_arr[] = $arr[$i];

}else{

$right\_arr[] = $arr[$i];

}

}

return array\_merge(quick\_sort($left\_arr),array($arr[0]),quick\_sort($right\_arr));

}

**适用场景**：

**稳定度**：意思是说原本键值一样的元素排序后相对位置不变。n为问题的规模，大写英文字母O为算法复杂度。

**冒泡排序**：当n问题的规模小时，对原本键值一样的元素排序后相对位置不变要求高时适用。

**选择排序**：当n问题的规模小时，对原本键值一样的元素排序后相对位置不变无要求时适用。

**插入排序**：在大部分已排序好的时候适用。

**快速排序**：当n问题的规模大时，对原本键值一样的元素排序后相对位置不变无要求时适用。



1. 二分查找 log2(n)

**条件**：二分查找法需要数组是**一个有序的数组**。

**步骤**：

1. 假设我们的数组是一个递增的数组，首先我们需要找到数组的中间位置。
2. 要知道中间位置就需要知道起始位置和结束位置，然后取出中间位置的值来和我们的值做对比。
3. 如果中间值大于我们的给定值，说明我们的值在中间位置之前，此时需要再次二分，因为在中间之前，所以我们需要变的值是结束位置的值，此时结束位置的值应该是我们此时的中间位置。
4. 反之，如果中间值小于我们给定的值，那么说明给定值在中间位置之后，此时需要再次将后一部分的值进行二分，因为在中间值之后，所以我们需要改变的值是开始位置的值，此时开始位置的值应该是我们此时的中间位置，直到我们找到指定值。
5. 或者中间值等于最初的起始位置，或结束位置（此时说明给定值未找到）。

//循环实现

function getValue($num,$arr){

//查找数组的中间位置

$length = count($arr);

$start = 0;

$end = $length;

$middle = floor(($start + $end) / 2);

//循环判断

while($start > $end - 1){

if($arr[middle] == $num){

return middle + 1;

}elseif($arr[middle] < $num){

//如果当前要查找的值比当前数组的中间值还要打，那么意味着该值在数组的后半段

//所以起始位置变成当前的middle的值，end位置不变。

$start = $middle;

$middle = floor(($start + $end) / 2);

}else{

//反之

$end = $middle;

$middle = floor(($start + $end) / 2);

}

}

return false;

}

//递归实现

/\*

\* 从数组中获取元素值

\* @param1 int $num，要查找的目标值

\* @param2 array $arr，要查找的数组

\* @param3 int $start，查找的起始位置

\* @param4 int $end，查找的结束位置

\* @return mixed，找到了返回位置，没找到返回false

\*/

function getValue4($num,$arr,$start = 0,$end = 100){

//采用二分法查找

$middle = floor(($end + $start) / 2);

//判断

if($arr[$middle] == $num){

//已经找到了，递归的出口

return $middle + 1;

}elseif($arr[$middle] < $num){

//要查找的元素在数组的后半段

$start = $middle + 1;

//边界值

if($start >= $end){

//没有找到，但是已经超出边界值，递归出口

return false;

}

//调用自己去查找：递归点

return getValue4($num,$arr,$start,$end); //getValue4($num,$arr,51,100)

}else{

//要查找的元素在数组的前半段

$end = $middle - 1;

//判断边界值

if($end < 0)return false;

//调用自己：递归点

return getValue4($num,$arr,$start,$end); //getValue4($num,$arr,0,49)

}

//都没有找到

return false;

}

1. 排列组合

function plzh($arr,$size=5) {

$len = count($arr);

$max = pow(2,$len);

$min = pow(2,$size)-1;

$r\_arr = array();

for ($i=$min; $i<$max; $i++){

$count = 0;

$t\_arr = array();

for ($j=0; $j<$len; $j++){

$a = pow(2, $j);

$t = $i&$a;

if($t == $a){

$t\_arr[] = $arr[$j];

$count++;

}

}

if($count == $size){

$r\_arr[] = $t\_arr;

}

}

return $r\_arr;

}

$pl = plzh(array(1,2,3,4,5,6,7),5);

var\_dump($pl);

1. 阶乘 O(log2n)

function jie\_cheng($n){

if($n<=1){

return 1;

}else{

return $n\*jc($n-1);

}

}

1. 简单意义上的桶排序

简单意义上的桶排序原理是先安排N+1个桶作为容器,若数据范围为N的话。 然后将测试数据(所需排序的数据)进行循环，放入对应的桶内。数据一定是在范围N内的。最后,循环桶里的元素,并且输出,进行从大到小或从小到大的排序。

**示例**:

（1）我们的取值范围是10,那么就要定义一个 11长度的数组$arr. 并且让所有的元素值都为0；

（2）然后,对需要排序的数组进行循环 如5,3,5,2,8；

（3）将数据依次对应$arr桶数组内元素,即如果是5,则使$arr[5]++；

（4）这时候 $arr[2]=1 $arr[3]=1 $arr[5]=2 $arr[8]=1；

（5）然后循环$arr的数组,若$arr[2]=1,则循环输出元素2一次,$arr[5]=2,则循环输出5两次；

 （6）结果输出即为 2 3 5 5 8；

（7）如果循环数值是从大到小 则会是从大到小的排序

//设置默认数组,默认值为0;

$arr = array();

for ($i = 0; $i <= 10; $i++) {

$arr[$i] = 0;

}

//设置测试的五个数据

$arr1 = array(5, 3, 5, 2, 8);

//根据数据 对默认数组的对应元素进行+1; J的取值范围不能等于$arr1数组长度

for ($j = 0; $j < count($arr1); $j++) {

//这边给相应的数组值+1

$arr[$arr1[$j]]++;

}

//开始循环输出 默认数组 $arr 里面相应的值

for ($k = 0; $k <= 10; $k++) {

for ($l=1; $l <=$arr[$k]; $l++) {

echo "$k </n>";

}

}

**缺点：**

（1）浪费空间；

（2）无法对浮点数据的排序。

1. MySQL数据库及优化方案
2. 索引

索引（Index）是帮助MySQL高效获取数据的数据结构；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，可以在这些数据结构上实现高级查找算法，提高查询速度，这种数据结构，就是索引。

索引是在MySQL的**存储引擎层中实现的**，而不是在服务层实现的。所以各种存储引擎支持的索引并不相同，MySQL目前提供了以下4种索引。

（1）**B-Tree 索引**：最常见的索引类型，大部分引擎都支持B树索引。  
（2）**HASH 索引**：只有Memory引擎支持，使用场景简单。  
（3） **R-Tree 索引(空间索引)**：空间索引是**MyISAM**的一种特殊索引类型，主要用于地理空间数据类型。  
（4）**Full-text (全文索引)**：全文索引也是**MyISAM**的一种特殊索引类型，主要用于全文索引，InnoDB从MySQL5.6版本提供对全文索引的支持。

* 1. **B-TREE索引类型**

1. **普通索引**

普通索引是最基本的索引类型，而且它没有唯一性之类的限制，可以通过以下几种方式创建：  
（1）创建索引: CREATE INDEX 索引名 ON 表名(列名1，列名2,…);  
（2）修改表: ALTER TABLE 表名 ADD INDEX 索引名 (列名1，列名2,…);  
（3）创建表时指定索引：CREATE TABLE 表名 ( […], INDEX 索引名 (列名1，列名 2,…) );

1. **Unique索引**

唯一索引表示唯一的，不允许重复的索引，若某一字段的信息不能重复（例如身份证号），可以将该字段的索引设置为unique：  
（1）创建索引：CREATE UNIQUE INDEX 索引名 ON 表名(列名1，列名2,…);  
（2）修改表：ALTER TABLE 表名ADD UNIQUE 索引名 (列名1，列名2,…);  
（3）创建表时指定索引：CREATE TABLE 表名( […], UNIQUE 索引名 (列名1，列名2,…));

1. **Primary Key索引**

主键是一种唯一性索引，但它必须指定为“PRIMARY KEY”。可以将其理解为 索引名固定为 PRIMARY KEY 的 UNIQUE索引。

（1）主键一般在创建表的时候指定：“CREATE TABLE 表名( […], PRIMARY KEY (列的列表) ); ”。  
（2）但是，我们也可以通过修改表的方式加入主键：“ALTER TABLE 表名 ADD PRIMARY KEY (列的列表); ”。  
**每个表只能有一个主键。（主键相当于聚合索引，是查找最快的索引）**  
注：**不能用CREATE INDEX语句创建PRIMARY KEY索引**

1. **主键索引与唯一索引的区别：**

主键是一种约束，唯一索引是一种索引，两者在本质上是不同的。

主键创建后一定包含一个唯一性索引，唯一性索引并不一定就是主键。

唯一性索引列允许空值，而主键列不允许为空值。

主键列在创建时，已经默认为非空值 + 唯一索引了。

主键可以被其他表引用为外键，而唯一索引不能。

一个表最多只能创建一个主键，但可以创建多个唯一索引。

~~主键和唯一索引都可以有多列。~~

索引可以提高查询的速度，而主键不能。

主键和索引都是键，不过主键是逻辑键，索引是物理键，意思就是主键不实际存在，而索引实际存在在数据库中，会占用磁盘。

主键一般都要建，主要是用来避免一张表中有相同的记录，索引一般可以不建，但如果需要对该表进行查询操作，则最好建，这样可以加快检索的速度。

主键更适合那些不容易更改的唯一标识，如自动递增列、身份证号等。

在 RBO 模式下，主键的执行计划优先级要高于唯一索引。 两者可以提高查询的速度。

1. **联合索引（复合索引）**

联合索引又叫复合索引，即一个覆盖表中两列或者以上的索引。

**原则：最左原则。**

经典命题：

假设某个表有一个联合索引（c1,c2,c3,c4）以下选项哪些字段使用了该索引：

A. where c1=x and c2=x and c4>x and c3=x

B. where c1=x and c2=x and c4=x order by c3

C. where c1=x and c4= x group by c3,c2

D. where c1=? and c5=? order by c2,c3

E. where c1=? and c2=? and c5=? order by c2,c3

通过MySQL explain SQL语句执行计划，查看索引的使用情况。

order by 和group by 类似，字段顺序与索引一致时，会使用索引排序；字段顺序与索引不一致时，不使用索引。

综上所述答案是：

A:四个字段均使用了该索引

B:c1，c2字段使用了该索引

C:c1字段使用该索引

D:c1字段使用该索引

E:c1，c2字段使用了该索引

**Explain SQL语句执行计划中的列：**

* **id列**：id列的编号是 select 的序列号，有几个 select 就有几个id，并且id的顺序是按 select 出现的顺序增长的。MySQL将 select 查询分为简单查询和复杂查询。复杂查询分为三类：简单子查询、派生表（from语句中的子查询）、union 查询。

简单子查询：explain select (select 1 from actor limit 1) from film;

from字句中的子查询：查询执行时有个临时表别名为der，外部 select 查询引用了这个临时表

union查询：union结果总是放在一个匿名临时表中，临时表不在SQL总出现，因此它的id是NULL。

* **select\_type列**：表示对应行是是简单还是复杂的查询，如果是复杂的查询，又是上述三种复杂查询中的哪一种。

simple：简单查询。查询不包含子查询和union；

primary：复杂查询中最外层的 select

subquery：包含在 select 中的子查询（不在 from 子句中）

derived：包含在 from 子句中的子查询。MySQL会将结果存放在一个临时表中，也称为派生表（derived的英文含义）

union：在 union 中的第二个和随后的 select

union result：从 union 临时表检索结果的 select

* **table列**：表示 explain 的一行正在访问哪个表。当 from 子句中有子查询时，table列是 <derivenN> 格式，表示当前查询依赖 id=N 的查询，于是先执行 id=N 的查询。当有 union 时，UNION RESULT 的 table 列的值为 <union1,2>，1和2表示参与 union 的 select 行id。
* **type列**：依次从最优到最差分别为system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > ALL

**NULL**：mysql能够在优化阶段分解查询语句，在执行阶段用不着再访问表或索引。例如：在索引列中选取最小值，可以单独查找索引来完成，不需要在执行时访问表

**const, system**：mysql能对查询的某部分进行优化并将其转化成一个常量（可以看show warnings 的结果）。用于 primary key 或 unique key 的所有列与常数比较时，所以表最多有一个匹配行，读取1次，速度比较快。

**eq\_ref**：primary key 或 unique key 索引的所有部分被连接使用 ，最多只会返回一条符合条件的记录。这可能是在 const 之外最好的联接类型了，简单的 select 查询不会出现这种 type。

**ref**：相比 eq\_ref，不使用唯一索引，而是使用普通索引或者唯一性索引的部分前缀，索引要和某个值相比较，可能会找到多个符合条件的行。

**ref\_or\_null**：类似ref，但是可以搜索值为NULL的行。

**index\_merge**：表示使用了索引合并的优化方法。 例如下表：id是主键，tenant\_id是普通索引。or 的时候没有用 primary key，而是使用了 primary key(id) 和 tenant\_id 索引

**range**：范围扫描通常出现在 in(), between ,> ,<, >= 等操作中。使用一个索引来检索给定范围的行。

**index**：和ALL一样，不同就是mysql只需扫描索引树，这通常比ALL快一些。

**ALL**：即全表扫描，意味着mysql需要从头到尾去查找所需要的行。通常情况下这需要增加索引来进行优化了

* **possible\_keys列**：显示查询可能使用哪些索引来查找。explain 时可能出现 possible\_keys 有列，而 key 显示 NULL 的情况，这种情况是因为表中数据不多，mysql认为索引对此查询帮助不大，选择了全表查询。 如果该列是NULL，则没有相关的索引。在这种情况下，可以通过检查 where 子句看是否可以创造一个适当的索引来提高查询性能，然后用 explain 查看效果。
* **key列**：显示mysql实际采用哪个索引来优化对该表的访问。如果没有使用索引，则该列是 NULL。如果想强制mysql使用或忽视possible\_keys列中的索引，在查询中使用 force index、ignore index。
* key\_len列：显示了mysql在索引里使用的字节数，通过这个值可以算出具体使用了索引中的哪些列。 举例来说，film\_actor的联合索引 idx\_film\_actor\_id 由 film\_id 和 actor\_id 两个int列组成，并且每个int是4字节。通过结果中的key\_len=4可推断出查询使用了第一个列：film\_id列来执行索引查找。

**key\_len计算规则如下：**

**字符串**

* char(n)：n字节长度
* varchar(n)：2字节存储字符串长度，如果是utf-8，则长度 3n + 2

**数值类型**

* tinyint：1字节
  + smallint：2字节
  + int：4字节
* bigint：8字节

**时间类型**

* date：3字节
* timestamp：4字节
  + datetime：8字节

如果字段允许为 NULL，需要1字节记录是否为 NULL

索引最大长度是**768字节**，当字符串过长时，mysql会做一个类似左前缀索引的处理，将前半部分的字符提取出来做索引。

* **ref列**：显示了在key列记录的索引中，表查找值所用到的列或常量，常见的有：const（常量），func，NULL，字段名（例：film.id）
* **rows列**：是mysql估计要读取并检测的行数，注意这个不是结果集里的行数。
* **Extra列**：展示的是额外信息。常见的重要值如下：

**distinct**: 一旦mysql找到了与行相联合匹配的行，就不再搜索了；

**Using index**：这发生在对表的请求列都是同一索引的部分的时候，返回的列数据只使用了索引中的信息，而没有再去访问表中的行记录。是性能高的表现；

**Using where**：mysql服务器将在存储引擎检索行后再进行过滤。就是先读取整行数据，再按 where 条件进行检查，符合就留下，不符合就丢弃。

**Using temporary**：mysql需要创建一张临时表来处理查询。出现这种情况一般是要进行优化的，首先是想到用索引来优化。

**Using filesort**：mysql 会对结果使用一个外部索引排序，而不是按索引次序从表里读取行。此时mysql会根据联接类型浏览所有符合条件的记录，并保存排序关键字和行指针，然后排序关键字并按顺序检索行信息。这种情况下一般也是要考虑使用索引来优化的。

* 1. **常用语法**

**设置索引** 在执行CREATE TABLE语句时可以创建索引，也可以单独用CREATE INDEX或ALTER TABLE来为数据表增加索引。

1. ALTER TABLE - ALTER TABLE可以用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。

ALTER TABLE table\_name ADD INDEX index\_name (column\_list)

ALTER TABLE table\_name ADD UNIQUE index\_name (column\_list)

ALTER TABLE table\_name ADD PRIMARY KEY (column\_list)

1. CREATE INDEX - CREATE INDEX可对表增加普通索引或UNIQUE索引。

CREATE INDEX index\_name ON table\_name (column\_list)

CREATE UNIQUE INDEX index\_name ON table\_name (column\_list)

**删除索引** 可利用ALTER TABLE或DROP INDEX语句来删除索引。类似于CREATE INDEX语句，DROP INDEX可以在ALTER TABLE内部作为一条语句处理，语法如下。

DROP INDEX index\_name ON talbe\_name

ALTER TABLE table\_name DROP INDEX index\_name

ALTER TABLE table\_name DROP PRIMARY KEY

其中，前两条语句是等价的，删除掉table\_name中名为index\_name的索引。第3条语句只在删除PRIMARY KEY索引时使用，因为一个表只可能有一个PRIMARY KEY索引，因此不需要指定索引名。如果没有创建PRIMARY KEY索引，但表具有一个或多个UNIQUE索引，则MySQL将删除第一个UNIQUE索引。

如果从表中删除了某列，则索引会受到影响。对于多列组合的索引，如果删除其中的某列，则该列也会从索引中删除。如果删除组成索引的所有列，则整个索引将被删除。

**查看索引**

mysql> show index from tblname;

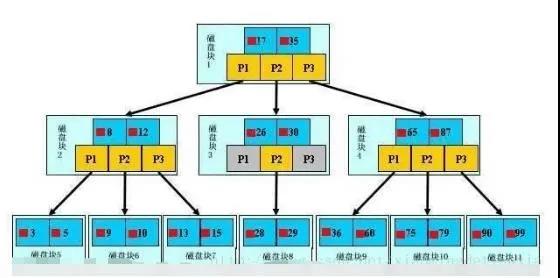
**索引设置的原则**

* 较频繁的作为查询条件的字段应该创建索引
* 唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件
* 更新非常频繁的字段不适合创建索引
* 不会出现在 WHERE 子句中的字段不该创建索引
* 索引的选择性较低不宜建索引

注：所谓索引的选择性（Selectivity），是指不重复的索引值（也叫基数，Cardinality）与表记录数的比值，显然选择性的取值范围为(0, 1]：

**索引的缺陷**  索引文件本身要消耗存储空间，同时索引会加重插入、删除和修改记录时的负担，另外，MySQL在运行时也要消耗资源维护索引，因此索引并不是越多越好。

* 1. **B+ 树**



如上图，是一颗B+树，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。**非叶子节点不存储真实的数据，只存储指引搜索方向的数据项**，如17、35并不真实存在于数据表中。

**查找过程**：

如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

**B+树性质**：

1. 通过上面的分析，我们知道IO次数取决于b+树的高度h，假设当前数据表的数据量为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么B+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。  
    2. 当B+树的数据项是复合的数据结构的时候，比如(name,age,sex)，B+树是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，B+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，B+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，B+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。
   1. **常见问题**

**1、什么是索引?**

索引是一种数据结构,可以帮助我们快速的进行数据的查找.

**2、索引是个什么样的数据结构呢?**

索引的数据结构和具体存储引擎的实现有关, 在MySQL中使用较多的索引有Hash索引,B+树索引等,而我们经常使用的InnoDB存储引擎的默认索引实现为:B+树索引.

**3、Hash索引和B+树所有有什么区别或者说优劣呢?**

首先要知道Hash索引和B+树索引的底层实现原理:

hash索引底层就是hash表,进行查找时,调用一次hash函数就可以获取到相应的键值,之后进行回表查询获得实际数据.B+树底层实现是多路平衡查找树.对于每一次的查询都是从根节点出发,查找到叶子节点方可以获得所查键值,然后根据查询判断是否需要回表查询数据.

那么可以看出他们有以下的不同:

hash索引进行等值查询更快(一般情况下),但是却无法进行范围查询.

因为在hash索引中经过hash函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询.而B+树的的所有节点皆遵循(左节点小于父节点,右节点大于父节点,多叉树也类似),天然支持范围.

hash索引不支持使用索引进行排序,原理同上.

hash索引不支持模糊查询以及多列索引的最左前缀匹配.原理也是因为hash函数的不可预测.AAAA和AAAAB的索引没有相关性.

hash索引任何时候都避免不了回表查询数据,而B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询.

hash索引虽然在等值查询上较快,但是不稳定.性能不可预测,当某个键值存在大量重复的时候,发生hash碰撞,此时效率可能极差.而B+树的查询效率比较稳定,对于所有的查询都是从根节点到叶子节点,且树的高度较低.

因此,在大多数情况下,直接选择B+树索引可以获得稳定且较好的查询速度.而不需要使用hash索引.

**4、上面提到了B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据,什么是聚簇索引?**

在B+树的索引中,叶子节点可能存储了当前的key值,也可能存储了当前的key值以及整行的数据,这就是聚簇索引和非聚簇索引. 在InnoDB中,只有主键索引是聚簇索引,如果没有主键,则挑选一个唯一键建立聚簇索引.如果没有唯一键,则隐式的生成一个键来建立聚簇索引.

当查询使用聚簇索引时,在对应的叶子节点,可以获取到整行数据,因此不用再次进行回表查询.

**5、非聚簇索引一定会回表查询吗?**

不一定,这涉及到查询语句所要求的字段是否全部命中了索引,如果全部命中了索引,那么就不必再进行回表查询.

举个简单的例子,假设我们在员工表的年龄上建立了索引,那么当进行select age from employee where age < 20的查询时,在索引的叶子节点上,已经包含了age信息,不会再次进行回表查询.

**6、在建立索引的时候,都有哪些需要考虑的因素**呢?

建立索引的时候一般要考虑到字段的使用频率,经常作为条件进行查询的字段比较适合.如果需要建立联合索引的话,还需要考虑联合索引中的顺序.此外也要考虑其他方面,比如防止过多的索引对表造成太大的压力.这些都和实际的表结构以及查询方式有关.

**7、联合索引是什么?为什么需要注意联合索引中的顺序?**

MySQL可以使用多个字段同时建立一个索引,叫做联合索引.在联合索引中,如果想要命中索引,需要按照建立索引时的字段顺序挨个使用,否则无法命中索引.

具体原因为:

MySQL使用索引时需要索引有序,假设现在建立了"name,age,school"的联合索引,那么索引的排序为: 先按照name排序,如果name相同,则按照age排序,如果age的值也相等,则按照school进行排序.

当进行查询时,此时索引仅仅按照name严格有序,因此必须首先使用name字段进行等值查询,之后对于匹配到的列而言,其按照age字段严格有序,此时可以使用age字段用做索引查找,以此类推.因此在建立联合索引的时候应该注意索引列的顺序,一般情况下,将查询需求频繁或者字段选择性高的列放在前面.此外可以根据特例的查询或者表结构进行单独的调整.

**8、创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?**

MySQL提供了explain命令来查看语句的执行计划,MySQL在执行某个语句之前,会将该语句过一遍查询优化器,之后会拿到对语句的分析,也就是执行计划,其中包含了许多信息. 可以通过其中和索引有关的信息来分析是否命中了索引,例如possilbe\_key,key,key\_len等字段,分别说明了此语句可能会使用的索引,实际使用的索引以及使用的索引长度.

**9、那么在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢?**

使用不等于查询；

列参与了数学运算或者函数；

在字符串like时左边是通配符.类似于'%aaa'；

当mysql分析全表扫描比使用索引快的时候不使用索引；

当使用联合索引,前面一个条件为范围查询,后面的即使符合最左前缀原则,也无法使用索引。

以上情况,MySQL无法使用索引.

1. 存储过程

存储过程（Stored Procedure）是一组为了完成某种特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

一个存储过程是一个可编程的函数，它在数据库中创建并保存。它可以有SQL语句和一些特殊的控制结构组成。当希望在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对面向对象方法的模拟，它允许控制数据的访问方式。

**优点**：

1. 存储过程增强了SQL语言的功能和灵活性：存储过程可以用流控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。  
   (2) 存储过程允许标准组件式编程：存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且可以随时对存储过程进行修改，对应用程序源代码毫无影响。  
   (3) 存储过程能实现较快的执行速度：如果某一操作包含大量的Transaction-SQL代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在每次运行时都要进行编译和优化，速度相对要慢一些。  
   (4) 存储过程能减少网络流量：针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织成存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，从而大大增加了网络流量并降低了网络负载。  
   (5) 存储过程可被作为一种安全机制来充分利用：系统管理员通过执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全。
2. 事务

**事务是并发控制的基本单位**。所谓的事务，它是一个操作序列，由一条或者多条sql语句组成，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。

**ACID特性**：

事务应该具有四个属性：原子性、一致性、隔离性、持久性。

**原子性（Atomicity）**：指整个数据库事务是不可分割的工作单位。只有事务中所有的数据库操作都执行成功，整个事务的执行才算成功。事务中任何一个sql语句执行失败，那么已经执行成功的sql语句也必须撤销，数据库状态应该退回到执行事务前的状态。  
 **一致性（Consistency）**：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态的含义是数据库中的数据应满足完整性约束，也就是说在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏  
 **隔离性（Isolation）**：隔离性也叫做并发控制、可串行化或者锁。事务的隔离性要求每个读写事务的对象与其它事务的操作对象能相互分离，即该事务提交前对其它事务都不可见，这通常使用锁来实现多个事务并发执行时，一个事务的执行不应影响其他事务的执行。  
 **持久性（Durability）**：表示事务一旦提交了，其结果就是永久性的，也就是数据就已经写入到数据库了，如果发生了宕机等事故，数据库也能将数据恢复。

**事务的分类**

一般分为扁平事务、带有保存点的扁平事务、链事务、嵌套事务、分布式事务。

**扁平事务**是最简单的一种，也是实际开发中使用的最多的一种事务。在这种事务中，所有操作都处于同一层次。扁平事务很简单，但有一个主要缺点是不能提交或回滚事务的某一部分，或者分几个独立的步骤去提交。

**带有保存点的扁平事务**这种事务除了支持扁平事务支持的操作外，允许在事务执行过程中回滚到同一事务中较早的一个状态，这是因为可能某些事务在执行过程中出现的错误并不会对所有的操作都无效，放弃整个事务不合乎要求，开销也太大。保存点用来通知系统应该记住事务当前的状态，以便以后发生错误时，事务能回到该状态。

**链事务**就是指回滚时，只能恢复到最近一个保存点；而带有保存点的扁平事务则可以回滚到任意正确的保存点。

**嵌套事务**在事务中再嵌套事务，位于根节点的事务称为顶层事务。事务的前驱称为父事务，事务的下一层称为子事务。子事务既可以提交也可以回滚，但是它的提交操作并不马上生效，除非由其父事务提交。因此就可以确定，任何子事务都在顶层事务提交后才真正的被提交了。同理，任意一个事务的回滚都会引起它的所有子事务一同回滚。

**分布式事务**通常是指在一个分布式环境下运行的扁平事务，因此需要根据数据所在位置访问网络中的不同节点，比如：通过建设银行向招商银行转账，建设银行和招商银行肯定用的不是同一个数据库，同时二者的数据库也不在一个网络节点上，那么当用户跨行转账，就是通过分布式事务来保证数据的ACID的。

**MySQL中使用事务**

在MySQL命令行的默认设置下，事务都是自动提交的，即执行SQL语句后就会马上执行COMMIT操作。因此要显示地开启一个事务须使用命令BEGIN或START TRANSACTION，或者执行命令SET AUTOCOMMIT=0，用来禁止使用当前会话的自动提交。

**事务控制语句**

**BEGIN或START TRANSACTION**；显示地开启一个事务；

**COMMIT**；也可以使用COMMIT WORK，不过二者是等价的。COMMIT会提交事务，并使已对数据库进行的所有修改称为永久性的；

**ROLLBACK**；有可以使用ROLLBACK WORK，不过二者是等价的。回滚会结束用户的事务，并撤销正在进行的所有未提交的修改；

**SAVEPOINT identifier**；SAVEPOINT允许在事务中创建一个保存点，一个事务中可以有多个SAVEPOINT；

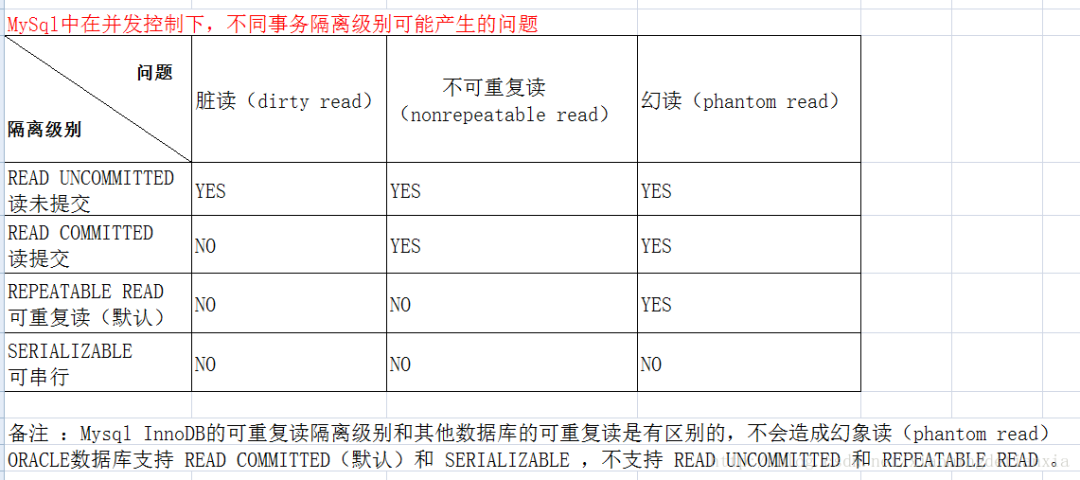
**RELEASE SAVEPOINT identifier**；删除一个事务的保存点，当没有指定的保存点时，执行该语句会抛出一个异常；

**ROLLBACK TO identifier**；把事务回滚到标记点；

**SET TRANSACTION**；用来设置事务的隔离级别。

**事务的隔离级别**

事务的隔离级别就是通过锁的机制来实现，**锁的应用最终导致不同事务的隔离级别**。在数据库操作中，为了有效保证并发读取数据的正确性，提出的事务隔离级别。InnoDB存储引擎提供事务的隔离级别有READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ和SERIALIZABLE。这些隔离级别之间的区别如下：



**脏读**Read uncommitted：一个事务读取到了另外一个事务没有提交的数据**；**  
比如：事务T1更新了一行记录的内容，但是并没有提交所做的修改。事务T2读取到了T1更新后的行，然后T1执行回滚操作，取消了刚才所做的修改。现在T2所读取的行就无效了；  
**不可重复**读Read committed：在同一事务中，两次读取同一数据，得到内容不同**；**一个事务读取到另外一个事务已经提交的数据，也就是说一个事务可以看到其他事务所做的修改**。**比如：事务T1读取一行记录，紧接着事务T2修改了T1刚才读取的那一行记录。然后T1又再次读取这行记录，发现与刚才读取的结果不同。这就称为“不可重复”读，因为T1原来读取的那行记录已经发生了变化；

**Repeatable read**：避免不可重复读是事务级别的快照！每次读取的都是当前事务的版本，即使被修改了，也只会读取当前事务版本的数据  
**幻读**：同一事务中，用同样的操作读取两次，得到的记录数不相同；是指在一个事务内读取到了别的事务插入的数据，导致前后读取不一致。和不可重复读类似，但虚读(幻读)会读到其他事务的插入的数据，导致前后读取不 一致，幻读的重点在于新增或者删除(数据条数变化)，不可重复读的重点是修改。  
比如：事务T1读取一条指定的WHERE子句所返回的结果集。然后事务T2新插入 一行记录，这行记录恰好可以满足T1所使用的查询条件中的WHERE子句的条件。然后T1又使用相同的查询再次对表进行检索，但是此时却看到了事务T2刚才插入的新行。这个新行就称为“幻像”，因为对T1来说这一行就像突然出现的一样。  
隔离级别越低，事务请求的锁越少或保持锁的时间就越短。InnoDB存储引擎默认的支持隔离级别是REPEATABLE READ；在这种默认的事务隔离级别下已经能完全保证事务的隔离性要求，即达到SQL标准的SERIALIZABLE级别隔离。

**设置事务的隔离级别**

SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL

{

READ UNCOMMITTED

| READ COMMITTED

| REPEATABLE READ

| SERIALIZABLE

}

**注意**：使用GLOBAL | SESSION  的话设置的是默认的隔离级别，注意default默认二字，GLOBAL将设置全局默认的隔离级别，而SESSION将设置本次会话(可以理解为你连接到mysql服务器的这个窗口，别人也会开窗口但是你影响不到别人的窗口中去)的所有事务的隔离级别并在当前这个事务中起效。

要GLOBAL | SESSION  都不用，就是设置本次命令及以后的隔离级别，注意不是默认隔离级别，如果执行了不带GLOBAL | SESSION  的set命令将设置了当前事务的隔离级别，但是如果下个事务没有使用set设置的话就会调用默认的隔离级别。

1. 视图

视图是一种虚拟的表，具有和物理表相同的功能，可以对视图进行增，改，查操作，视图通常是有一个表或者多个表的行或列的子集，对视图的修改不影响基本表，它使得我们获取数据更容易，相比多表查询。

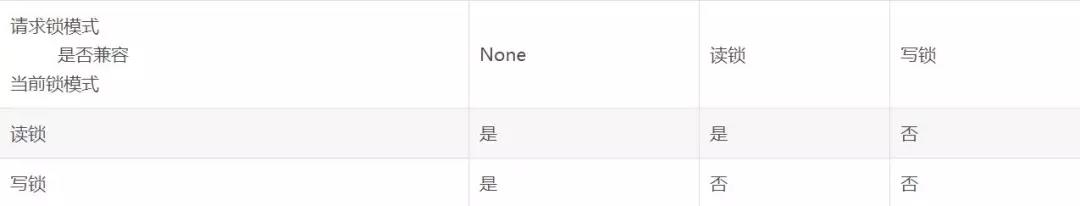
1. 超键 候选键 主键 外键

**超键**：在关系中能唯一标识元组(数据库中的一条记录)的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。  
 **候选键**：是最小超键，即没有冗余元素的超键。  
 **主键**：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合，用户选作元组标识的一个侯选键称为主键。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。  
 **外键**：在一个表中存在的另一个表的主键称此表的外键，外键主要是用来描述两个表的关系。

1. 表锁与行锁
   1. **表锁**

从锁的粒度，我们可以分成两大类：  
 **表锁 ：**开销小，加锁快；不会出现死锁；锁定力度大，发生锁冲突概率高，并发度最低；  
 **行锁：**开销大，加锁慢；会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高不同的存储引擎支持的锁粒度是不一样的。InnoDB行锁和表锁都支持、MyISAM只支持表锁！InnoDB只有通过索引条件检索数据才使用行级锁，否则，InnoDB使用表锁也就是说，InnoDB的行锁是基于索引的！

**表锁下又分为两种模式**： 表读锁（Table Read Lock）&& 表写锁（Table Write Lock）从下图可以清晰看到，在表读锁和表写锁的环境下：读读不阻塞，读写阻塞，写写阻塞！读读不阻塞：当前用户在读数据，其他的用户也在读数据，不会加锁 读写阻塞：当前用户在读数据，其他的用户不能修改当前用户读的数据，会加锁！写写阻塞：当前用户在修改数据，其他的用户不能修改当前用户正在修改的数据，会加锁！

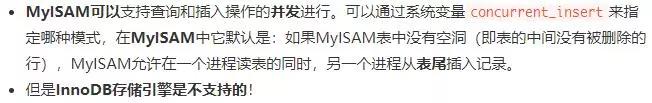


从上面已经看到了：读锁和写锁是互斥的，读写操作是串行。

如果某个进程想要获取读锁，同时另外一个进程想要获取写锁。在mysql中，写锁是优先于读锁的！

写锁和读锁优先级的问题是可以通过参数调节的：max\_write\_lock\_count和low-priority-updates

**注意：**

****

* 1. **行锁**

InnoDB和MyISAM有两个**本质的区别**：InnoDB支持行锁、InnoDB支持事务。

**InnoDB实现了以下两种类型的行锁**：

**共享锁（S锁、读锁）**：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。即多个客户可以同时读取同一个资源，但不允许其他客户修改。

**排他锁（X锁、写锁)：**允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的读锁和写锁。写锁是排他的，写锁会阻塞其他的写锁和读锁。

另外，为了允许行锁和表锁共存，实现多粒度锁机制，**InnoDB还有两种内部使用的意向锁（Intention Locks）**，这两种意向锁都是表锁：

**意向共享锁（IS）：**事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。

**意向排他锁（IX）：**事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。

**MVCC**

**MVCC(Multi-Version ConcurrencyControl)多版本并发控制**，可以简单地认为：MVCC就是行级锁的一个变种(升级版)。在表锁中我们读写是阻塞的，基于提升并发性能的考虑，MVCC一般读写是不阻塞的(很多情况下避免了加锁的操作)。可以简单的理解为：对数据库的任何修改的提交都不会直接覆盖之前的数据，而是产生一个新的版本与老版本共存，使得读取时可以完全不加锁。

* 1. **间隙锁**

当我们用范围条件检索数据而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合范围条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”。InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁。例子：假如emp表中只有101条记录，其empid的值分别是1,2,...,100,101

Select \* from emp where empid > 100 for update;

上面是一个范围查询，InnoDB不仅会对符合条件的empid值为101的记录加锁，也会对empid大于101（这些记录并不存在）的“间隙”加锁

**InnoDB使用间隙锁的目的有2个：**

为了防止幻读(上面也说了，Repeatable read隔离级别下再通过GAP锁即可避免了幻读)

满足恢复和复制的需要：MySQL的恢复机制要求在一个事务未提交前，其他并发事务不能插入满足其锁定条件的任何记录，也就是不允许出现幻读

* 1. **死锁**

死锁是指两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。表级锁不会产生死锁.所以解决死锁主要还是针对于最常用的InnoDB。

死锁的关键在于：两个(或以上)的Session加锁的顺序不一致。

对应的解决死锁问题的关键就是：让不同的session加锁有次序。

当对存在的行进行锁的时候(主键)，mysql就只有行锁。

当对未存在的行进行锁的时候(即使条件为主键)，mysql是会锁住一段范围（有gap锁）。

insert语句对于主键来说，插入的行不管有没有存在，都会只有行锁

* 1. **总结**

在MyISAM存储引擎中，当执行SQL语句的时候是自动加的。

在InnoDB存储引擎中，如果没有使用索引，表锁也是自动加的。

现在我们大多数使用MySQL都是使用InnoDB，InnoDB支持行锁：

共享锁--读锁--S锁

排它锁--写锁--X锁

在默认的情况下，select是不加任何行锁的~事务可以通过以下语句显示给记录集加共享锁或排他锁。

共享锁（S）：SELECT \* FROM table\_name WHERE ... LOCK IN SHARE MODE

排他锁（X)：SELECT \* FROM table\_name WHERE ... FOR UPDATE

InnoDB基于行锁还实现了MVCC多版本并发控制，MVCC在隔离级别下的Read committed和Repeatable read下工作。MVCC实现了读写不阻塞。

1. 乐观锁与悲观锁

数据库管理系统（DBMS）中的并发控制的任务是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和一致性以及数据库的统一性。

乐观并发控制(乐观锁)和悲观并发控制（悲观锁）是并发控制采用的主要技术手段。

无论是悲观锁还是乐观锁，都是一种概念，是一种思想。其实不仅仅是关系型数据库系统中有乐观锁和悲观锁的概念，像memcache、hibernate、tair等都有类似的概念。针对不同的业务场景，应该选用不同的并发控制方式。所以，不要把乐观并发控制和悲观并发控制狭义的理解为仅在DBMS中存在的概念，更不要把他们和数据库中提供的锁机制（行锁、表锁、排他锁、共享锁）混为一谈。其实，在DBMS中，悲观锁正是利用数据库本身提供的锁机制来实现的。

**悲观锁**

在关系数据库管理系统里，悲观并发控制（又名“悲观锁”，Pessimistic Concurrency Control，缩写“PCC”）是一种并发控制的方法。它可以阻止一个事务以影响其他用户的方式来修改数据。如果一个事务执行的操作在某行数据上应用了锁，那只有当这个事务把锁释放，其他事务才能够执行与该锁冲突的操作。

悲观并发控制主要用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本的环境中。

**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(悲观)，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）。

**悲观锁的流程：**  
 1. 在对某一记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。  
 2. 如果加锁失败，说明该记录正在被修改，那么当前操作可能要等待或者抛出异常， 具体响应方式由开发者根据实际情况决定。  
 3. 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。  
 4. 其间如果有其他事务要对该记录做修改或加排他锁，都会等待该事务将该记录解锁或直接抛出异常。

**MySQL InnoDB中使用悲观锁**

**注意**：要使用悲观锁，必须先关闭mysql数据库的自动提交功能，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。

set autocommit=0;

**示例：**

//0.开始事务

begin;/begin work;/start transaction; (三者选一就可以)

//1.查询出商品信息

select status from t\_goods where id=1 for update;

//2.根据商品信息生成订单

insert into t\_orders (id,goods\_id) values (null,1);

//3.修改商品status为2

update t\_goods set status=2;

//4.提交事务

commit;/commit work;

上面的查询语句中，我们使用了select…for update的方式，这样就**通过开启排他锁的方式实现了悲观锁**。此时在t\_goods表中，id为1的 那条数据就被我们锁定了，其它事务必须等本次事务提交之后才能对该记录进行操作。这样我们可以保证当前的数据不会被其它事务修改。

**注意**：上面提到，使用select…for update会把数据给锁住，不过我们需要注意一下锁的级别，MySQL **InnoDB默认为行级锁**。**行级锁都是基于索引的，如果一条SQL语句没有用到索引是不会使用行级锁的，会使用表级锁把整张表锁住，**这点需要注意。

**优点与不足：**  
**优点**：悲观并发控制实际上是采用“先取锁再访问”的保守策略，为数据处理的安全性提供了保证；  
**缺点**：在效率方面，处理加锁的机制会让数据库产生额外的开销，同时会增加产生死锁的机率；另外，在只读型事务中由于不会产生冲突，也没必要使用锁，这样做只会增加系统负载；还会降低并行性，一个事务如果锁定了某行数据，其他事务就必须等待该事务处理完才可以处理那行数。

**乐观锁**

在关系数据库管理系统里，乐观并发控制（又名“乐观锁”，Optimistic Concurrency Control，缩写“OCC”）是一种并发控制的方法。它假设多用户并发的事务在处理数据时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务对该数据做过修改。如果其他事务更新过该数据的话，正在提交的事务会进行回滚。乐观事务控制最早是由孔祥重（H.T.Kung）教授提出。乐观锁（ Optimistic Locking ）是相对悲观锁而言，乐观锁假设数据一般情况下不会造成冲突，所以在事务对数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则返回错误信息，让用户决定如何去做。相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制，一般用记录数据版本的方式实现乐观锁。

**数据版本**：为数据增加的一个版本标识。当读取数据时，将版本标识的值一同读出，数据每更新一次，便对版本标识进行一次更新。当事务提交更新的时候，需要判断数据库表对应记录的当前版本信息与第一次取出来的版本标识是否一致，如果数据库表当前版本号与第一次取出来的版本标识值相等，则予以更新，否则认为是过期数据。

实现数据版本有两种方式，第一种是使用版本号，第二种是使用时间戳。

**使用版本号实现乐观锁**

使用版本号时，可以在数据初始化时指定一个版本号，每次对数据的更新操作都对版本号执行+1操作。并判断当前版本号是不是该数据的最新的版本号。

1.查询出商品信息

select (status,status,version) from t\_goods where id=#{id}

2.根据商品信息生成订单

3.修改商品status为2

update t\_goods

set status=2,version=version+1

where id=#{id} and version=#{version};

乐观并发控制假设事务之间的数据竞争(data race)概率比较小，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。但如果直接简单这么做，还是有可能会遇到不可预期的结果，例如两个事务都读取了数据库的某一行，经过修改以后写回数据库，这时就遇到了问题。

1. 存储引擎（MyISAM与InnoDB）
2. **存储结构**

**MyISAM**：每个MyISAM表在磁盘上存储成三个文件，文件的名字以表的名字开始，扩展名指出文件类型：.frm文件存储表定义；数据文件的扩展名为.MYD (MYData)；索引文件的扩展名是.MYI (MYIndex)。  
**InnoDB：**所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。

1. **存储空间**

**MyISAM**：可被压缩，存储空间较小。支持三种不同的存储格式：静态表(默认，但是注意数据末尾不能有空格，会被去掉)、动态表、压缩表。  
**InnoDB：**需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

1. **可移植性、备份及恢复**

**MyISAM**：数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。  
**InnoDB：**免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。

1. **事务支持**

**MyISAM**：强调的是性能，每次查询具有原子性,其执行数度比InnoDB类型更快，但是不提供事务支持。  
**InnoDB：**支持事务，外部键等高级数据库功能。具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。

事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而MyISAM就不可以了。

1. **AUTO\_INCREMENT**

**MyISAM**:可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。  
**InnoDB:** InnoDB中必须包含只有该字段的索引。引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

1. **表锁差异**

**MyISAM**:可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。  
**InnoDB:** InnoDB中必须包含只有该字段的索引。引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

1. **全文索引**

**MyISAM**：支持 FULLTEXT类型的全文索引。  
**InnoDB：**不支持FULLTEXT类型的全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。

1. **表主键**

**MyISAM**：允许没有任何索引和主键的表存在，索引都是保存行的地址。  
**InnoDB：**如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。

1. **表的具体行数**

**MyISAM**：保存有表的总行数，如果select count() from table;会直接取出出该值。  
**InnoDB：**没有保存表的总行数，如果使用select count() from table；就会遍历整个表，消耗相当大，但是在加了wehre条件后，**MyISAM**和innodb处理的方式都一样。

1. **外键**

**MyISAM**：不支持  
**InnoDB：**支持  
通过上述的分析，基本上可以考虑使用InnoDB来替代**MyISAM**引擎了，原因是InnoDB自身很多良好的特点，比如事务支持、存储 过程、视图、行级锁定等等，在并发很多的情况下，相信InnoDB的表现肯定要比**MyISAM**强很多。另外，任何一种表都不是万能的，只用恰当的针对业务类型来选择合适的表类型，才能最大的发挥MySQL的性能优势。如果不是很复杂的Web应用，非关键应用，还是可以继续考虑**MyISAM**的，这个具体视情况而定。

1. **CURD操作**

**MyISAM**：如果执行大量的SELECT，**MyISAM**是更好的选择。  
**InnoDB：**如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。DELETE 从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table这个命令。

1. 设计范式

**第一范式（1NF）**：数据库表中的字段都是单一属性的，不可再分。这个单一属性由基本类型构成，包括整型、实数、字符型、逻辑型、日期型等。

**第二范式（2NF）**：数据库表中不存在非关键字段对任一候选关键字段的部分函数依赖（部分函数依赖指的是存在组合关键字中的某些字段决定非关键字段的情况），也即所有非关键字段都完全依赖于任意一组候选关键字。  
 **第三范式（3NF）**：在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖则符合第三范式。所谓传递函数依赖，指的是如 果存在”A → B → C”的决定关系，则C传递函数依赖于A。因此，满足第三范式的数据库表应该不存在如下依赖关系：关键字段 → 非关键字段 x → 非关键字段y。

1. 主从复制
2. SQL注入问题
3. 分库 分表 分区

开发中经常会遇到大表的情况，所谓的大表是指存储了百万级乃至千万级条记录的表。这样的表过于庞大，导致数据库在查询和写入的时候耗时太长，性能低下，如果涉及联合查询的情况，性能会更加糟糕。分表和表分区的**目的就是**减少数据库的负担，提高数据库的效率，通常点来讲就是提高表的增删改查效率。

**数据库瓶颈**

不管是IO瓶颈还是CPU瓶颈，最终都会导致数据库的活跃连接数增加，进而逼近甚至达到数据库可承载的活跃连接数的阈值。在业务service来看， 就是可用数据库连接少甚至无连接可用，接下来就可以想象了（并发量、吞吐量、崩溃）。

**IO瓶颈**

第一种：磁盘读IO瓶颈，热点数据太多，数据库缓存放不下，每次查询会产生大量的IO，降低查询速度 分库和垂直分表

第二种：网络IO瓶颈，请求的数据太多，网络带宽不够 分库

**CPU瓶颈**

第一种：SQL问题：如SQL中包含join,group by, order by，非索引字段条件查询等，增加CPU运算的操作 SQL优化，建立合适的索引，在业务Service层进行业务计算。

第二种：单表数据量太大，查询时扫描的行太多，SQL效率低，增加CPU运算的操作。 水平分表。

**分表**是将一个大表按照一定的规则分解成多张具有独立存储空间的实体表，我们可以称为子表，每个表都对应三个文件，MYD数据文件，.MYI索引文件，.frm表结构文件。这些子表可以分布在同一块磁盘上，也可以在不同的机器上。app读写的时候根据事先定义好的规则得到对应的子表名，然后去操作它。

在分表之前，首先要选中合适的分表策略（以哪个字典为分表字段，需要将数据分为多少张表），使数据能够均衡的分布在多张表中，并且不影响正常的查询。在企业级应用中，往往使用org\_id(组织主键)做为分表字段，在互联网应用中往往是userid。在确定分表策略后，当数据进行存储及查询时，需要确定到哪张表里去查找数据，

 数据存放的数据表 = 分表字段的内容 % 分表数量

**分库** 分表能够解决单表数据量过大带来的查询效率下降的问题，但是不能给数据库的并发访问带来质的提升，面对高并发的写访问，当Master无法承担高并发的写入请求时，不管如何扩展Slave服务器，都没有意义了。我们通过对数据库进行拆分，来提高数据库的写入能力，即所谓的分库。分库采用对关键字取模的方式，对数据库进行路由。

数据存放的数据库 = 分库字段的内容 % 数据库的数量

**分区**和分表相似，都是按照规则分解表。不同在于分表将大表分解为若干个独立的实体表，而分区是将数据分段划分在多个位置存放，可以是同一块磁盘也可以在不同的机器。分区后，表面上还是一张表，但数据散列到多个位置了。app读写的时候操作的还是大表名字，db自动去组织分区的数据。

**既分表又分库** 数据库分表可以解决单表海量数据的查询性能问题，分库可以解决单台数据库的并发访问压力问题。当数据库同时面临海量数据存储和高并发访问的时候，需要同时采取分表和分库策略。一般分表分库策略如下：

中间变量 = 关键字%（数据库数量\*单库数据表数量）

库 = 取整（中间变量/单库数据表数量）

表 = （中间变量%单库数据表数量）

**分区与分表的联系**

1. 都能提高mysql的性高，在高并发状态下都有一个良好的表现。  
   2. 分表和分区不矛盾，可以相互配合的，对于那些大访问量，并且表数据比较多的表，我们可以采取分表和分区结合的方式（如果merge这种分表方式，不能和分区配合的话，可以用其他的分表试），访问量不大，但是表数据很多的表，我们可以采取分区的方式等。  
   3. 分表技术是比较麻烦的，需要手动去创建子表，app服务端读写时候需要计算子表名。采用merge好一些，但也要创建子表和配置子表间的union关系。  
   4. 分区相对于分表，操作方便，不需要创建子表。

* **水平分库**

水平分库是指以字段为依据，按照一定策略（hash、range等），将一个库中的数据拆分到多个库中。其结果是：每个库的结构都一样；每个库中的数据不一样，没有交集；所有库的数据并集是全量数据。主要应用于系统绝对并发量上来了，分表难以根本上解决问题，并且还没有明显的业务归属来垂直分库的情况下。这样库多了，io和cpu的压力自然可以成倍缓解。

* **垂直分库**

垂直分库是以表为依据，按照业务归属不同，将不同的表拆分到不同的库中。其结果是每个库的结构都不一样；每个库的数据也不一样，没有交集；所有库的并集是全量数据。主要应用于系统绝对并发量上来了，并且可以抽象出单独的业务模块的情况下。到这一步，基本上就可以服务化了。例如：随着业务的发展，一些公用的配置表、字典表等越来越多，这时可以将这些表拆到单独的库中，甚至可以服务化。再者，随着业务的发展孵化出了一套业务模式，这时可以将相关的表拆到单独的库中，甚至可以服务化。

* **水平分表**

水平分表是以字段为依据，按照一定策略（hash、range等），讲一个表中的数据拆分到多个表中。其结果是每个表的结构都一样；每个表的数据不一样，没有交集，所有表的并集是全量数据。主要应用于系统绝对并发量没有上来，只是单表的数据量太多，影响了SQL效率，加重了CPU负担，以至于成为瓶颈，可以考虑水平分表。这样单表的数据量少了，单次执行SQL执行效率高了，自然减轻了CPU的负担。

* **垂直分表**

垂直分表是以字段为依据，按照字段的活跃性，将表中字段拆到不同的表中（主表和扩展表）。其结果是每个表的结构不一样；每个表的数据也不一样，一般来说，每个表的字段至少有一列交集，一般是主键，用于关联数据；所有表的并集是全量数据。主要应用于系统绝对并发量并没有上来，表的记录并不多，但是字段多，并且热点数据和非热点数据在一起，单行数据所需的存储空间较大，以至于数据库缓存的数据行减少，查询时回去读磁盘数据产生大量随机读IO，产生IO瓶颈。

垂直分表可以用列表页和详情页来帮助理解。垂直分表的拆分原则是将热点数据（可能经常会查询的数据）放在一起作为主表，非热点数据放在一起作为扩展表，这样更多的热点数据就能被缓存下来，进而减少了随机读IO。拆了之后，要想获取全部数据就需要关联两个表来取数据。但记住千万别用join，因为Join不仅会增加CPU负担并且会将两个表耦合在一起（必须在一个数据库实例上）。关联数据应该在service层进行，分别获取主表和扩展表的数据，然后用关联字段关联得到全部数据。

* **水平分区**

mysql数据库中的数据是以文件的形势存在磁盘上的，默认放在/mysql/data下面（可以通过my.cnf中的datadir来查看），一张表主要对应着三个文件，一个是frm存放表结构的，一个是myd存放表数据的，一个是myi存表索引的。如果一张表的数据量太大的话，那么myd,myi就会变的很大，查找数据就会变的很慢，这个时候我们可以利用mysql的分区功能，在物理上将这一张表对应的三个文件，分割成许多个小块，这样呢，我们查找一条数据时，就不用全部查找了，只要知道这条数据在哪一块，然后在那一块找就行了。如果表的数据太大，可能一个磁盘放不下，这个时候，我们可以把数据分配到不同的磁盘里面去。

举例来说明一下，假如有100W条数据，分成十份，前10W条数据放到第一个分区，第二个10W条数据放到第二个分区，依此类推。也就是把表分成了十分，根用merge来分表，有点像哦。取出一条数据的时候，这条数据包含了表结构中的所有字段，也就是说横向分区，并没有改变表的结构。

**横向分区的几种方式**

1. **MySQL5.1及以上支持分区功能**

mysql> show variables like "%part%";

1. **Range分区**

这种模式允许将数据划分不同范围。例如可以将一个表通过年份划分成若干个分区。

create table t\_range(

　　 id int(11),

　　 money int(11) unsigned not null,

　　 date datetime

　　)partition by range(year(date))(

　　partition p2007 values less than (2008),

　　partition p2008 values less than (2009),

　　partition p2009 values less than (2010)

　　partition p2010 values less than maxvalue #MAXVALUE 表示最大的可能的整数值

　　)；

RANGE分区在如下场合特别有用：

1）、当需要删除一个分区上的“旧的”数据时,只删除分区即可。如果你使用上面最近的

那个例子给出的分区方案，你只需简单地使用”ALTER TABLE employees DROP PARTITION p0;”

来删除所有在1991年前就已经停止工作的雇员相对应的所有行。对于有大量行的表，

这比运行一个如”DELETE FROM employees WHERE YEAR (separated) <= 1990;”

　　这样的一个DELETE查询要有效得多。

2）、想要使用一个包含有日期或时间值，或包含有从一些其他级数开始增长的值的列。

3）、经常运行直接依赖于用于分割表的列的查询。

　　例如，当执行一个如”SELECT COUNT(\*) FROM employees WHERE YEAR(separated) = 2000 GROUP BY store\_id;”这样的查询时，

　　MySQL可以很迅速地确定只有分区p2需要扫描，这是因为余下的分区不可能包含有符合该WHERE子句的任何记录

1. **List分区**

这种模式允许系统通过预定义的列表的值来对数据进行分割。

create table t\_list(

　　a int(11),

　　b int(11)

　　)(partition by list (b)

　　partition p0 values in (1,3,5,7,9),

　　partition p1 values in (2,4,6,8,0)

);

LIST分区没有类似如“VALUES LESS THAN MAXVALUE”这样的包含其他值在内的定义。

将要匹配的任何值都必须在值列表中找到。

1. **Hash分区**

这中模式允许通过对表的一个或多个列的Hash Key进行计算，最后通过这个Hash码不同数值对应的数据区域进行分区。例如可以建立一个对表主键进行分区的表。

CREATE TABLE employees (

id INT NOT NULL,

fname VARCHAR(30),

lname VARCHAR(30),

hired DATE NOT NULL DEFAULT '1970-01-01',

separated DATE NOT NULL DEFAULT '9999-12-31',

job\_code INT,

store\_id INT

)

PARTITION BY HASH(store\_id)

PARTITIONS 4;

1. **Key分区**

上面Hash模式的一种延伸，这里的Hash Key是MySQL系统产生的。

CREATE TABLE tk (

col1 INT NOT NULL,

col2 CHAR(5),

col3 DATE

)

PARTITION BY LINEAR KEY (col1)

PARTITIONS 3;

1. **子分区**

子分区是分区表中每个分区的再次分割，子分区既可以使用HASH希分区，也可以使用KEY分区。这也被称为复合分区（composite partitioning）。

1，如果一个分区中创建了子分区，其他分区也要有子分区

2，如果创建了了分区，每个分区中的子分区数必有相同

3，同一分区内的子分区，名字不相同，不同分区内的子分区名子可以相同（5.1.50不适用）

mysql> CREATE TABLE IF NOT EXISTS `sub\_part` (

-> `news\_id` int(11) NOT NULL COMMENT '新闻ID',

-> `content` varchar(1000) NOT NULL DEFAULT '' COMMENT '新闻内容',

-> `u\_id` int(11) NOT NULL DEFAULT 0s COMMENT '来源IP',

-> `create\_time` DATE NOT NULL DEFAULT '0000-00-00 00:00:00' COMMENT '时间'

-> ) ENGINE=INNODB DEFAULT CHARSET=utf8

-> PARTITION BY RANGE(YEAR(create\_time))

-> SUBPARTITION BY HASH(TO\_DAYS(create\_time))(

-> PARTITION p0 VALUES LESS THAN (1990)(SUBPARTITION s0,SUBPARTITION s1,SUBPARTITION s2),

-> PARTITION p1 VALUES LESS THAN (2000)(SUBPARTITION s3,SUBPARTITION s4,SUBPARTITION good),

-> PARTITION p2 VALUES LESS THAN MAXVALUE(SUBPARTITION tank0,SUBPARTITION tank1,SUBPARTITION tank3)

-> );

Query OK, 0 rows affected (0.07 sec)

**分区的优点：**

1，分区可以分在多个磁盘，存储更大一点；

2，根据查找条件，也就是where后面的条件，查找只查找相应的分区不用全部查找了；

3，进行大数据搜索时可以进行并行处理；

4，跨多个磁盘来分散数据查询，来获得更大的查询吞吐量。

1. **分区管理**

**删除分区**

alter table user drop partion p4

**新增分区**

alter table user add partition(partition p4 values less than MAXVALUE)；#新增range分区

alter table list\_part add partition(partition p4 values in(25,26,27)) #新增list分区

alter table hash\_part add partition partitions 4; # hash重新分区

alter table key\_part add partition partitions 4; #key 重新分区

//子分区添加新分区，虽然我没有指定子分区，但是系统会给子分区命名的

alter table sub1\_part add partition(partition p3 values less than MAXVALUE);

//range重新分区

ALTER TABLE user REORGANIZE PARTITION p0,p1,p2,p3,p4 INTO (PARTITION p0 VALUES LESS THAN MAXVALUE);

//list重新分区

ALTER TABLE list\_part REORGANIZE PARTITION p0,p1,p2,p3,p4 INTO (PARTITION p0 VALUES in (1,2,3,4,5));

#hash和key分区不能用REORGANIZE，官方网站说的很清楚

* **垂直分区**

举例来说明，在设计用户表的时候，开始的时候没有考虑好，而把个人的所有信息都放到了一张表里面去，这样这个表里面就会有比较大的字段，如个人简介，而这些简介呢，也许不会有好多人去看，所以等到有人要看的时候，在去查找，分表的时候，可以把这样的大字段，分开来。

* **分表的几种方式**

1. **MySQL集群**

事实它并不是分表，但起到了和分表相同的作用。集群可分担数据库的操作次数，将任务分担到多台数据库上。集群可以读写分离，减少读写压力。从而提升数据库性能。

1. **自定义规则分表**

大表可以按照业务的规则来分解为多个子表。通常为以下几种类型，也可自己定义规则。

1. Range（范围）–这种模式允许将数据划分不同范围。例如可以将一个表通过年份划分成若干个分区。
2. Hash（哈希）–这中模式允许通过对表的一个或多个列的Hash Key进行计算，最后通过这个Hash码不同数值对应的数据区域进行分区。例如可以建立一个对表主键进行分区的表。

（3） Key（键值）-上面Hash模式的一种延伸，这里的Hash Key是MySQL系统产生的。

（4） List（预定义列表）–这种模式允许系统通过预定义的列表的值来对数据进行分割。

（5） composite（复合模式） –以上模式的组合使用。

**优点**：避免一张表出现几百万条数据，缩短了一条sql的执行时间。

**缺陷**：当一种规则确定时，打破这条规则会很麻烦，上面的例子中我用的hash算法是crc32，如果我现在不想用这个算法了，改用md5后，会使同一个用户的消息被存储到不同的表中，这样数据乱套了。扩展性很差。

1. **利用merge存储引擎实现分表**

**优点：**扩展性好，并且程序代码改动的不是很大。

**缺陷：**这种方法的效果比第二种要差一点。

* **带来的问题**

1. **事务一致性问题**

**分布式事务**

当更新内容同时分布在不同库中，不可避免会带来跨库事务问题。跨分片事务也是分布式事务，没有简单的方案，一般可使用"XA协议"和"两阶段提交"处理。

分布式事务能最大限度保证了数据库操作的原子性。但在提交事务时需要协调多个节点，推后了提交事务的时间点，延长了事务的执行时间。导致事务在访问共享资源时发生冲突或死锁的概率增高。随着数据库节点的增多，这种趋势会越来越严重，从而成为系统在数据库层面上水平扩展的枷锁。

**最终一致性**

对于那些性能要求很高，但对一致性要求不高的系统，往往不苛求系统的实时一致性，只要在允许的时间段内达到最终一致性即可，可采用事务补偿的方式。与事务在执行中发生错误后立即回滚的方式不同，事务补偿是一种事后检查补救的措施，一些常见的实现方法有：对数据进行对账检查，基于日志进行对比，定期同标准数据来源进行同步等等。事务补偿还要结合业务系统来考虑。

1. **跨节点关联查询join问题**

切分之前，系统中很多列表和详情页所需的数据可以通过sql join来完成。而切分之后，数据可能分布在不同的节点上，此时join带来的问题就比较麻烦了，考虑到性能，尽量避免使用join查询。

**解决方法：**

1. **全局表**

全局表，也可看做是"数据字典表"，就是系统中所有模块都可能依赖的一些表，为了避免跨库join查询，可以将这类表在每个数据库中都保存一份。这些数据通常很少会进行修改，所以也不担心一致性的问题。

1. **字段冗余**

一种典型的反范式设计，利用空间换时间，为了性能而避免join查询。例如：订单表保存userId时候，也将userName冗余保存一份，这样查询订单详情时就不需要再去查询"买家user表"了。

但这种方法适用场景也有限，比较适用于依赖字段比较少的情况。而冗余字段的数据一致性也较难保证，就像上面订单表的例子，买家修改了userName后，是否需要在历史订单中同步更新呢？这也要结合实际业务场景进行考虑。

1. **数据组装**

在系统层面，分两次查询，第一次查询的结果集中找出关联数据id，然后根据id发起第二次请求得到关联数据。最后将获得到的数据进行字段拼装。

1. **ER分片**

关系型数据库中，如果可以先确定表之间的关联关系，并将那些存在关联关系的表记录存放在同一个分片上，那么就能较好的避免跨分片join问题。在1:1或1:n的情况下，通常按照主表的ID主键切分。

1. **跨节点分页、排序、函数问题**

跨节点多库进行查询时，会出现limit分页、order by排序等问题。分页需要按照指定字段进行排序，当排序字段就是分片字段时，通过分片规则就比较容易定位到指定的分片；当排序字段非分片字段时，就变得比较复杂了。需要先在不同的分片节点中将数据进行排序并返回，然后将不同分片返回的结果集进行汇总和再次排序，最终返回给用户。

1. **全局主键避重问题**

在分库分表环境中，由于表中数据同时存在不同数据库中，主键值平时使用的自增长将无用武之地，某个分区数据库自生成的ID无法保证全局唯一。因此需要单独设计全局主键，以避免跨库主键重复问题。

**解决方法：**

1. **UUID**

UUID标准形式包含32个16进制数字，分为5段，形式为8-4-4-4-12的36个字符，例如：550e8400-e29b-41d4-a716-446655440000。

UUID是主键是最简单的方案，本地生成，性能高，没有网络耗时。但缺点也很明显，由于UUID非常长，会占用大量的存储空间；另外，作为主键建立索引和基于索引进行查询时都会存在性能问题，在InnoDB下，UUID的无序性会引起数据位置频繁变动，导致分页。

1. **结合数据库维护主键ID表**

在数据库中建立 sequence 表：

CREATE TABLE `sequence` (

`id` bigint(20) unsigned NOT NULL auto\_increment,

`stub` char(1) NOT NULL default '',

PRIMARY KEY (`id`),

UNIQUE KEY `stub` (`stub`)

) ENGINE=MyISAM;

stub字段设置为唯一索引，同一stub值在sequence表中只有一条记录，可以同时为多张表生成全局ID。sequence表的内容，如下所示：

+-------------------+------+

| id | stub |

+-------------------+------+

| 72157623227190423 | a |

+-------------------+------+

使用 MyISAM 存储引擎而不是 InnoDB，以获取更高的性能。MyISAM使用的是表级别的锁，对表的读写是串行的，所以不用担心在并发时两次读取同一个ID值。

当需要全局唯一的64位ID时，执行：

REPLACE INTO sequence (stub) VALUES ('a');

SELECT LAST\_INSERT\_ID();

这两条语句是Connection级别的，select last\_insert\_id() 必须与 replace into 在同一数据库连接下才能得到刚刚插入的新ID。

使用replace into代替insert into**好处**是避免了表行数过大，不需要另外定期清理。

此方案较为简单，但**缺点**也明显：存在单点问题，强依赖DB，当DB异常时，整个系统都不可用。配置主从可以增加可用性，但当主库挂了，主从切换时，数据一致性在特殊情况下难以保证。另外性能瓶颈限制在单台MySQL的读写性能。

1. **Snowflake分布式自增ID算法**

Twitter的snowflake算法解决了分布式系统生成全局ID的需求，生成64位的Long型数字，组成部分：

·第一位未使用；

·接下来41位是毫秒级时间，41位的长度可以表示69年的时间；

·5位datacenterId，5位workerId。10位的长度最多支持部署1024个节点；

·最后12位是毫秒内的计数，12位的计数顺序号支持每个节点每毫秒产生4096个ID序列。

**好处**：毫秒数在高位，生成的ID整体上按时间趋势递增；不依赖第三方系统，稳定性和效率较高，理论上QPS约为409.6w/s（1000\*2^12），并且整个分布式系统内不会产生ID碰撞；可根据自身业务灵活分配bit位。

**不足**：强依赖机器时钟，如果时钟回拨，则可能导致生成ID重复。

1. **数据迁移、扩容问题**

当业务高速发展，面临性能和存储的瓶颈时，才会考虑分片设计，此时就不可避免的需要考虑历史数据迁移的问题。一般做法是先读出历史数据，然后按指定的分片规则再将数据写入到各个分片节点中。此外还需要根据当前的数据量和QPS，以及业务发展的速度，进行容量规划，推算出大概需要多少分片（一般建议单个分片上的单表数据量不超过1000W）

如果采用数值范围分片，只需要添加节点就可以进行扩容了，不需要对分片数据迁移。如果采用的是数值取模分片，则考虑后期的扩容问题就相对比较麻烦。

* **切分的时机**

1. **能不切分就不切分**

并不是所有表都需要进行切分，主要还是看数据的增长速度。切分后会在某种程度上提升业务的复杂度，数据库除了承载数据的存储和查询外，协助业务更好的实现需求也是其重要工作之一。

不到万不得已不用轻易使用分库分表这个大招，避免"过度设计"和"过早优化"。分库分表之前，不要为分而分，先尽力去做力所能及的事情，例如：升级硬件、升级网络、读写分离、索引优化等等。当数据量达到单表的瓶颈时候，再考虑分库分表。

1. **数据量过大，正常运维影响业务访问**
2. 对数据库备份，如果单表太大，备份时需要大量的磁盘IO和网络IO。例如1T的数据，网络传输占50MB时候，需要20000秒才能传输完毕，整个过程的风险都是比较高的
3. 对一个很大的表进行DDL修改时，MySQL会锁住全表，这个时间会很长，这段时间业务不能访问此表，影响很大。如果使用pt-online-schema-change，使用过程中会创建触发器和影子表，也需要很长的时间。在此操作过程中，都算为风险时间。将数据表拆分，总量减少，有助于降低这个风险。
4. 大表会经常访问与更新，就更有可能出现锁等待。将数据切分，用空间换时间，变相降低访问压力。
5. **随着业务发展，需要对某些字段垂直拆分**

在项目初始阶段，这种设计是满足简单的业务需求的，也方便快速迭代开发。而当业务快速发展时，用户量从10w激增到10亿，用户非常的活跃，每次登录会更新 last\_login\_name 字段，使得 user 表被不断update，压力很大。而其他字段：id, name, personal\_info 是不变的或很少更新的，此时在业务角度，就要将 last\_login\_time 拆分出去，新建一个 user\_time 表。

personal\_info 属性是更新和查询频率较低的，并且text字段占据了太多的空间。这时候，就要对此垂直拆分出 user\_ext 表了。

1. **数据量快速增长**

随着业务的快速发展，单表中的数据量会持续增长，当性能接近瓶颈时，就需要考虑水平切分，做分库分表了。此时一定要选择合适的切分规则，提前预估好数据容量。

1. **安全和可用性**

鸡蛋不要放在一个篮子里。在业务层面上垂直切分，将不相关的业务的数据库分隔，因为每个业务的数据量、访问量都不同，不能因为一个业务把数据库搞挂而牵连到其他业务。利用水平切分，当一个数据库出现问题时，不会影响到100%的用户，每个库只承担业务的一部分数据，这样整体的可用性就能提高。

* **分库分表工具**

sharding-jdbc（当当）、TSharding（蘑菇街）、Atlas（奇虎360）、Cobar（阿里巴巴）、MyCAT（基于Cobar）、Oceanus（58同城）、Vitess（谷歌）

1. **优化警句52条**

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

2，应尽量避免在 where 子句中对字段进行 null 值判断，创建表时NULL是默认值，但大多数时候应该使用NOT NULL，或者使用一个特殊的值，如0，-1作为默 认值。

3，应尽量避免在 where 子句中使用!=或<>操作符， MySQL只有对以下操作符才使用索引：<，<=，=，>，>=，BETWEEN，IN，以及某些时候的LIKE。

4，应尽量避免在 where 子句中使用 or 来连接条件， 否则将导致引擎放弃使用索引而进行全表扫描， 可以 使用UNION合并查询：select id from t where num=10 union all select id from t where num=20

5，in 和 not in 也要慎用，否则会导致全表扫描，对于连续的数值，能用 between 就不要用 in 了：Select id from t where num between 1 and 3

6，下面的查询也将导致全表扫描：select id from t where name like ‘%abc%’ 或者select id from t where name like ‘%abc’若要提高效率，可以考虑全文检索。而select id from t where name like ‘abc%’ 才用到索引

7， 如果在 where 子句中使用参数，也会导致全表扫描。

8，应尽量避免在 where 子句中对字段进行表达式操作，应尽量避免在where子句中对字段进行函数操作

9,很多时候用 exists 代替 in 是一个好的选择：select num from a where num in(select num from b).用下面的语句替换：select num from a where exists(select 1 from b where num=a.num)

10,索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

11,应尽可能的避免更新 clustered 索引数据列， 因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

12，尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。

13，尽可能的使用 varchar/nvarchar 代替 char/nchar ， 因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

14，最好不要使用”“返回所有：select from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

15，尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

16，使用表的别名(Alias)：当在SQL语句中连接多个表时,请使用表的别名并把别名前缀于每个Column上.这样一来,就可以减少解析的时间并减少那些由Column歧义引起的语法错误。

17，使用“临时表”暂存中间结果   
简化SQL语句的重要方法就是采用临时表暂存中间结果，但是，临时表的好处远远不止这些，将临时结果暂存在临时表，后面的查询就在tempdb中了，这可以避免程序中多次扫描主表，也大大减少了程序执行中“共享锁”阻塞“更新锁”，减少了阻塞，提高了并发性能。

18，一些SQL查询语句应加上nolock，读、写是会相互阻塞的，为了提高并发性能，对于一些查询，可以加上nolock，这样读的时候可以允许写，但缺点是可能读到未提交的脏数据。使用 nolock有3条原则。查询的结果用于“插、删、改”的不能加nolock ！查询的表属于频繁发生页分裂的，慎用nolock ！使用临时表一样可以保存“数据前影”，起到类似Oracle的undo表空间的功能，能采用临时表提高并发性能的，不要用nolock 。

19，常见的简化规则如下：不要有超过5个以上的表连接（JOIN），考虑使用临时表或表变量存放中间结果。少用子查询，视图嵌套不要过深,一般视图嵌套不要超过2个为宜。

20，将需要查询的结果预先计算好放在表中，查询的时候再Select。这在SQL7.0以前是最重要的手段。例如医院的住院费计算。

21，用OR的字句可以分解成多个查询，并且通过UNION 连接多个查询。他们的速度只同是否使用索引有关,如果查询需要用到联合索引，用UNION all执行的效率更高.多个OR的字句没有用到索引，改写成UNION的形式再试图与索引匹配。一个关键的问题是否用到索引。

22，在IN后面值的列表中，将出现最频繁的值放在最前面，出现得最少的放在最后面，减少判断的次数。

23，尽量将数据的处理工作放在服务器上，减少网络的开销，如使用存储过程。存储过程是编译好、优化过、并且被组织到一个执行规划里、且存储在数据库中的SQL语句，是控制流语言的集合，速度当然快。反复执行的动态SQL,可以使用临时存储过程，该过程（临时表）被放在Tempdb中。

24，当服务器的内存够多时，配制线程数量 = 最大连接数+5，这样能发挥最大的效率；否则使用 配制线程数量<最大连接数启用SQL SERVER的线程池来解决,如果还是数量 = 最大连接数+5，严重的损害服务器的性能。

25，查询的关联同写的顺序   
select a.personMemberID, \* from chineseresume a,personmember b where personMemberID = b.referenceid and a.personMemberID = ‘JCNPRH39681’ （A = B ,B = ‘号码’）   
select a.personMemberID, \* from chineseresume a,personmember b where a.personMemberID = b.referenceid and a.personMemberID = ‘JCNPRH39681’ and b.referenceid = ‘JCNPRH39681’ （A = B ,B = ‘号码’， A = ‘号码’）   
select a.personMemberID, \* from chineseresume a,personmember b where b.referenceid = ‘JCNPRH39681’ and a.personMemberID = ‘JCNPRH39681’ （B = ‘号码’， A = ‘号码’）

26，尽量使用exists代替select count(1)来判断是否存在记录，count函数只有在统计表中所有行数时使用，而且count(1)比count(\*)更有效率。

27，尽量使用“>=”，不要使用“>”。

28，索引的使用规范：索引的创建要与应用结合考虑，建议大的OLTP表不要超过6个索引；尽可能的使用索引字段作为查询条件，尤其是聚簇索引，必要时可以通过index index\_name来强制指定索引；避免对大表查询时进行table scan，必要时考虑新建索引；在使用索引字段作为条件时，如果该索引是联合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用；要注意索引的维护，周期性重建索引，重新编译存储过程。

29，下列SQL条件语句中的列都建有恰当的索引，但执行速度却非常慢：  
SELECT \* FROM record WHERE substrINg(card\_no,1,4)=’5378’ (13秒)   
SELECT \* FROM record WHERE amount/30< 1000 （11秒）   
SELECT \* FROM record WHERE convert(char(10),date,112)=’19991201’ （10秒）   
分析：  
WHERE子句中对列的任何操作结果都是在SQL运行时逐列计算得到的，因此它不得不进行表搜索，而没有使用该列上面的索引；如果这些结果在查询编译时就能得到，那么就可以被SQL优化器优化，使用索引，避免表搜索，因此将SQL重写成下面这样：  
SELECT \* FROM record WHERE card\_no like ‘5378%’ （< 1秒）   
SELECT \* FROM record WHERE amount< 1000\*30 （< 1秒）   
SELECT \* FROM record WHERE date= ‘1999/12/01’ （< 1秒）

30，当有一批处理的插入或更新时，用批量插入或批量更新，绝不会一条条记录的去更新!

31，在所有的存储过程中，能够用SQL语句的，我绝不会用循环去实现!   
(例如：列出上个月的每一天，我会用connect by去递归查询一下，绝不会去用循环从上个月第一天到最后一天)

32，选择最有效率的表名顺序(只在基于规则的优化器中有效)：  
oracle 的解析器按照从右到左的顺序处理FROM子句中的表名，FROM子句中写在最后的表(基础表 driving table)将被最先处理，在FROM子句中包含多个表的情况下,你必须选择记录条数最少的表作为基础表。如果有3个以上的表连接查询, 那就需要选择交叉表(intersection table)作为基础表, 交叉表是指那个被其他表所引用的表.

33，提高GROUP BY语句的效率, 可以通过将不需要的记录在GROUP BY 之前过滤掉.下面两个查询返回相同结果，但第二个明显就快了许多.   
低效:   
SELECT JOB , AVG(SAL)   
FROM EMP   
GROUP BY JOB   
HAVING JOB =’PRESIDENT’   
OR JOB =’MANAGER’   
高效:   
SELECT JOB , AVG(SAL)   
FROM EMP   
WHERE JOB =’PRESIDENT’   
OR JOB =’MANAGER’   
GROUP BY JOB

34，sql语句用大写，因为oracle 总是先解析sql语句，把小写的字母转换成大写的再执行。

35，别名的使用，别名是大型数据库的应用技巧，就是表名、列名在查询中以一个字母为别名，查询速度要比建连接表快1.5倍。

36，避免死锁，在你的存储过程和触发器中访问同一个表时总是以相同的顺序;事务应经可能地缩短，在一个事务中应尽可能减少涉及到的数据量;永远不要在事务中等待用户输入。

37，避免使用临时表，除非却有需要，否则应尽量避免使用临时表，相反，可以使用表变量代替;大多数时候(99%)，表变量驻扎在内存中，因此速度比临时表更快，临时表驻扎在TempDb数据库中，因此临时表上的操作需要跨数据库通信，速度自然慢。

38，最好不要使用触发器，触发一个触发器，执行一个触发器事件本身就是一个耗费资源的过程;如果能够使用约束实现的，尽量不要使用触发器;不要为不同的触发事件(Insert，Update和Delete)使用相同的触发器;不要在触发器中使用事务型代码。

39，索引创建规则：  
表的主键、外键必须有索引；  
数据量超过300的表应该有索引；  
经常与其他表进行连接的表，在连接字段上应该建立索引；  
经常出现在Where子句中的字段，特别是大表的字段，应该建立索引；  
索引应该建在选择性高的字段上；  
索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引；  
复合索引的建立需要进行仔细分析，尽量考虑用单字段索引代替；  
正确选择复合索引中的主列字段，一般是选择性较好的字段；  
复合索引的几个字段是否经常同时以AND方式出现在Where子句中？单字段查询是否极少甚至没有？如果是，则可以建立复合索引；否则考虑单字段索引；  
如果复合索引中包含的字段经常单独出现在Where子句中，则分解为多个单字段索引；  
如果复合索引所包含的字段超过3个，那么仔细考虑其必要性，考虑减少复合的字段；  
如果既有单字段索引，又有这几个字段上的复合索引，一般可以删除复合索引；  
频繁进行数据操作的表，不要建立太多的索引；  
删除无用的索引，避免对执行计划造成负面影响；  
表上建立的每个索引都会增加存储开销，索引对于插入、删除、更新操作也会增加处理上的开销。另外，过多的复合索引，在有单字段索引的情况下，一般都是没有存在价值的；相反，还会降低数据增加删除时的性能，特别是对频繁更新的表来说，负面影响更大。  
尽量不要对数据库中某个含有大量重复的值的字段建立索引。

40，mysql查询优化总结：使用慢查询日志去发现慢查询，使用执行计划去判断查询是否正常运行，总是去测试你的查询看看是否他们运行在最佳状态下。久而久之性能总会变化，避免在整个表上使用count(\*),它可能锁住整张表，使查询保持一致以便后续相似的查询可以使用查询缓存  
，在适当的情形下使用GROUP BY而不是DISTINCT，在WHERE, GROUP BY和ORDER BY子句中使用有索引的列，保持索引简单,不在多个索引中包含同一个列，有时候MySQL会使用错误的索引,对于这种情况使用USE INDEX，检查使用SQL\_MODE=STRICT的问题，对于记录数小于5的索引字段，在UNION的时候使用LIMIT不是是用OR。  
为了 避免在更新前SELECT，使用INSERT ON DUPLICATE KEY或者INSERT IGNORE ,不要用UPDATE去实现，不要使用 MAX,使用索引字段和ORDER BY子句，LIMIT M，N实际上可以减缓查询在某些情况下，有节制地使用，在WHERE子句中使用UNION代替子查询，在重新启动的MySQL，记得来温暖你的数据库，以确保您的数据在内存和查询速度快，考虑持久连接，而不是多个连接，以减少开销，基准查询，包括使用服务器上的负载，有时一个简单的查询可以影响其他查询，当负载增加您的服务器上，使用SHOW PROCESSLIST查看慢的和有问题的查询，在开发环境中产生的镜像数据中 测试的所有可疑的查询。

41，MySQL 备份过程:   
从二级复制服务器上进行备份。在进行备份期间停止复制，以避免在数据依赖和外键约束上出现不一致。彻底停止MySQL，从数据库文件进行备份。  
如果使用 MySQL dump进行备份，请同时备份二进制日志文件 – 确保复制没有中断。不要信任LVM 快照，这很可能产生数据不一致，将来会给你带来麻烦。为了更容易进行单表恢复，以表为单位导出数据 – 如果数据是与其他表隔离的。  
当使用mysqldump时请使用 –opt。在备份之前检查和优化表。为了更快的进行导入，在导入时临时禁用外键约束。  
为了更快的进行导入，在导入时临时禁用唯一性检测。在每一次备份后计算数据库，表以及索引的尺寸，以便更够监控数据尺寸的增长。  
通过自动调度脚本监控复制实例的错误和延迟。定期执行备份。

42，查询缓冲并不自动处理空格，因此，在写SQL语句时，应尽量减少空格的使用，尤其是在SQL首和尾的空格(因为，查询缓冲并不自动截取首尾空格)。

43，member用mid做標準進行分表方便查询么？一般的业务需求中基本上都是以username为查询依据，正常应当是username做hash取模来分表吧。分表的话 mysql 的partition功能就是干这个的，对代码是透明的；  
在代码层面去实现貌似是不合理的。

44，我们应该为数据库里的每张表都设置一个ID做为其主键，而且最好的是一个INT型的（推荐使用UNSIGNED），并设置上自动增加的AUTO\_INCREMENT标志。

45，在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。  
无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

46，MySQL查询可以启用高速查询缓存。这是提高数据库性能的有效Mysql优化方法之一。当同一个查询被执行多次时，从缓存中提取数据和直接从数据库中返回数据快很多。

47，EXPLAIN SELECT 查询用来跟踪查看效果   
使用 EXPLAIN 关键字可以让你知道MySQL是如何处理你的SQL语句的。这可以帮你分析你的查询语句或是表结构的性能瓶颈。EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的……等等，等等。

48，当只要一行数据时使用 LIMIT 1   
当你查询表的有些时候，你已经知道结果只会有一条结果，但因为你可能需要去fetch游标，或是你也许会去检查返回的记录数。在这种情况下，加上 LIMIT 1 可以增加性能。这样一样，MySQL数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据。

49,选择表合适存储引擎：  
myisam: 应用时以读和插入操作为主，只有少量的更新和删除，并且对事务的完整性，并发性要求不是很高的。  
Innodb：事务处理，以及并发条件下要求数据的一致性。除了插入和查询外，包括很多的更新和删除。（Innodb有效地降低删除和更新导致的锁定）。对于支持事务的InnoDB类型的表来说，影响速度的主要原因是AUTOCOMMIT默认设置是打开的，而且程序没有显式调用BEGIN 开始事务，导致每插入一条都自动提交，严重影响了速度。可以在执行sql前调用begin，多条sql形成一个事物（即使autocommit打开也可以），将大大提高性能。

50,优化表的数据类型,选择合适的数据类型：  
原则：更小通常更好，简单就好，所有字段都得有默认值,尽量避免null。  
例如：数据库表设计时候更小的占磁盘空间尽可能使用更小的整数类型.(mediumint就比int更合适)  
比如时间字段：datetime和timestamp, datetime占用8个字节，而timestamp占用4个字节，只用了一半，而timestamp表示的范围是1970—2037适合做更新时间  
MySQL可以很好的支持大数据量的存取，但是一般说来，数据库中的表越小，在它上面执行的查询也就会越快。  
因此，在创建表的时候，为了获得更好的性能，我们可以将表中字段的宽度设得尽可能小。例如，  
在定义邮政编码这个字段时，如果将其设置为CHAR(255),显然给数据库增加了不必要的空间，   
甚至使用VARCHAR这种类型也是多余的，因为CHAR(6)就可以很好的完成任务了。同样的，如果可以的话，  
我们应该使用MEDIUMINT而不是BIGIN来定义整型字段。  
应该尽量把字段设置为NOT NULL，这样在将来执行查询的时候，数据库不用去比较NULL值。  
对于某些文本字段，例如“省份”或者“性别”，我们可以将它们定义为ENUM类型。因为在MySQL中，ENUM类型被当作数值型数据来处理，  
而数值型数据被处理起来的速度要比文本类型快得多。这样，我们又可以提高数据库的性能。

51， 字符串数据类型：char，varchar，text选择区别

52，任何对列的操作都将导致表扫描，它包括数据库函数、计算表达式等等，查询时要尽可能将操作移至等号右边。

1. Redis
2. 数据类型

Redis是一个基于BSD开源的项目，是一个把结构化的数据放在内存中的一个存储系统，你可以把它作为数据库，缓存和消息中间件来使用。同时支持strings，lists，hashes，sets，sorted sets，bitmap，GeoHash，HyperLogLog，Streams（5.0以后）。

1. 数据持久化与恢复

Redis数据持久化策略主要分为RDB与AOF。由于 RDB 的数据实时性问题，目前用 AOF 比较多了，而持久化恢复也是优先 AOF。

**RDB数据持久化**

1. RDB 是一种快照模式，即保存的是 key value数据内容
2. RDB 有 2 种持久方式，同步 save 模式和异步 bgsave 模式。由于 save 是同步的，所以可以保证数据一致性，而 bgsave 则不能。
3. save 可以在客户端显式触发，也可以在 shutdown 时自动触发；bgsave 可以在客户端显式触发，也可以通过配置由定时任务触发，也可以在 slave 节点触发。
4. save 导致 redis 同步阻塞，基本已经废弃。bgsave 则不会导致阻塞，但也有缺点：在 fork 时，需要增加内存服务器开销，因为当内存不够时，将使用虚拟内存，导致阻塞 Redis 运行。所以，需要保证空闲内存足够。
5. 默认执行 shutdown 时，如果没有开启 AOF，则自动执行 bgsave。
6. 每次的 RDB 文件都是替换的。

**关于优化**：Redis 会压缩 RDB 文件，使用 **LZF 算法**，让最终的 RDB 文件远小于内存大小，默认开启。但会消耗 CPU。

**优点**：文件紧凑，适合备份，全量复制场景。例如每 6 小时执行 bgsave，保存到文件系统之类的。Redis 加载 RDB 恢复数据远远快于 AOF。

**缺点**：无法秒级持久化；老版本Redis无法兼容新版本RDB。

**AOF数据持久化**

由于 RDB 的数据实时性问题，AOF（append only file） 是目前 Redis 持久化的主流方式。

**特点：**

1、默认文件名是 appendonly.aof。和 RDB 一样，保存在配置中 dir 目录下。

2、AOF 相比较于 RDB，每次都会保存写命令，数据实时性更高。

3、AOF 由于每次都会记录写命令，文件会很大，因此需要进行优化，称之为“**重写机制**”

4、AOF 每次保存的写命令都放在一个缓冲区，根据不同的策略同步到磁盘。

**“重写机制”**：

1、fork 子进程（类似 bgsave）；

2、主进程会写到2个缓冲区，一个是原有的 “AOF 缓存区”，一个是专门为子进程准备的 “AOF 重写缓冲区”；

3、子进程写到到新的 AOF 文件中，批量的，默认 32m；写完后通知主进程；

4、主进程把“AOF 重写缓冲区”的数据写到新 AOF 文件中；

5、将新的 AOF 文件替换老文件。

**缓冲区同步策略，由参数 appendfsync 控制，一共3种：**

1、always：调用系统 fsync 函数，直到同步到硬盘返回；严重影响redis性能。

2、everysec：先调用 OS write 函数，写到缓冲区，然后 redis 每秒执行一次 OS fsync 函数。推荐使用这种方式。

3、no: 只执行 write OS 函数，具体同步硬盘策略由 OS 决定；不推荐，数据不安全，容易丢失数据。

**持久化恢复**

AOF 和 RDB 文件都可以用于服务器重启时的数据恢复。优先加载 AOF，当没有 AOF 时才加载 RDB。当 AOF 或者 RDB 存在错误，则加载失败。

1. **Fork操作问题及优化**

当 Redis做RDB或者AOF重写时，必然要进行 fork 操作，对于OS来说，fork 都是一个重量级操作。而且，fork 还会拷贝一些数据，虽然不会拷贝主进程所有的物理空间，但会复制主进程的空间内存页表。对于 10GB 的 Redis 进程，需要复制大约 20MB 的内存页表，因此 fork 操作耗时跟进程总内存量息息相关，再加上，如果使用虚拟化技术，例如 Xen 虚拟机，fork 会更加耗时。

一个正常的 fork 耗时大概在 20毫秒左右。为什么呢，假设一个 Redis 实例的 OPS 在 5 万以上，如果 fork 操作耗时在秒级，那么僵拖慢几万条命令的执行，对生产环境影响明显。

我们可以在 Info stats 统计中查询 latestforkusec 指标获取最近一次 fork 操作耗时，单位微秒。

**优化策略：**

1) 优先使用物理机或者高效支持 fork 的虚拟化技术，避免使用 Xen。

2)  控制 redis 实例最大内存，尽量控制在 10GB 以内。

3)  合理配置 Linux 内存分配策略，避免内存不足导致 fork 失败。

4)  降低 fork 的频率，如适度放宽 AOF 自动触发时机，避免不必要的全量复制。

1. **子进程开销问题**

fork 完毕之后，会创建子进程，子进程负责 RDB 或者 AOF 重写，这部分过程主要涉及到 CPU，内存，硬盘三个地方的优化。

1. CPU 写入文件的过程是 CPU 密集的过程，通常子进程对单核 CPU 利用率接近 90%。如何优化呢？既然是 CPU 密集型操作，就不要绑定单核 CPU，因为这样会和父 CPU 进行竞争。同时，不要和其他 CPU 密集型服务不是在一个机器上。如果部署了多个 Redis 实例，尽力保证统一时刻只有一个子进程执行重写工作。

2)  内存子进程通过 fork 操作产生，占用内存大小等同于父进程，理论上需要两倍的内存完成持久化操作，但 Linux 有 copy on write 机制，父子进程会共享相同的物理内存页，当父进程处理写操作时，会把要修改的页创建对应的副本，而子进程在 fork 操作过程中，共享整个父进程内存快照。即——如果重写过程中存在内存修改操作，父进程负责创建所修改内存页的副本。这里就是内存消耗的地方。如何优化呢？尽量保证同一时刻只有一个子进程在工作；避免大量写入时做重写操作。

3)  硬盘硬盘开销分析：子进程主要职责是将 RDB 或者 AOF 文件写入硬盘进行持久化，势必对硬盘造成压力，可通过工具例如 iostat，iotop 等，分析硬盘负载情况。

**优化策略：**

1) 不要和其他高硬盘负载的服务放在一台机器上，例如 MQ，存储。

2) AOF 重写时会消耗大量硬盘 IO，可以开启配置 no-appendfsync-on-rewrite，默认关闭。表示在 AOF 重写期间不做 fsync 操作。

3) 当开启 AOF 的 Redis 在高并发场景下，如果使用普通机械硬盘，每秒的写速率是 100MB左右，这时，Redis 的性能瓶颈在硬盘上，建议使用 SSD。

4) 对于单机配置多个 Redis 实例的情况，可以配置不同实例分盘存储 AOF 文件，分摊硬盘压力。

1. **AOF追加阻塞**

当开启 AOF 持久化时，常用的同步硬盘的策略是“每秒同步” everysec，用于平衡性能和数据安全性，对于这种方式，redis 使用另一条线程每秒执行 fsync 同步硬盘，当系统资源繁忙时，将造成 Redis 主线程阻塞。

**问题定位：**

1)  发生 AOF 阻塞时，会输入日志。用于记录 AOF fsync 阻塞导致拖慢 Redis 服务的行为。

2)  每当 AOF 追加阻塞事件发生时，在 info Persistence 统计中，aofdelayedfsync 指标会累加，查看这个指标方便定位 AOF 阻塞问题。

3)  AOF 同步最多运行 2 秒的延迟，当延迟发生时说明硬盘存在性能问题，可通过监控工具 iotop 查看，定位消耗 IO 的进程。

1. **单机多实例部署**

Redis 单线程架构无法充分利用多核CPU，通常的做法是一台机器上部署多个实例，当多个实例开启 AOF 后，彼此之间就会产生CPU 和 IO 的竞争。

让所有实例的 AOF 串行执行。通过 info Persistence 中关于 AOF 的信息写出 Shell 脚本，然后串行执行实例的 AOF 持久化。通过不断判断 AOF 的状态，手动执行 AOF 重写，保证 AOF 不会存在竞争。

1. 内存策略

Redis是基于内存的key-value数据库，因为系统的内存大小有限，所以我们在使用Redis的时候可以配置Redis能使用的最大的内存大小。

**通过配置文件修改内存大小**

//设置Redis最大占用内存大小为100M

maxmemory 100mb

**通过命令修改**

//设置Redis最大占用内存大小为100M

127.0.0.1:6379> config set maxmemory 100mb

//获取设置的Redis能使用的最大内存大小

127.0.0.1:6379> config get maxmemory

如果不设置最大内存大小或者设置最大内存大小为0，在64位操作系统下不限制内存大小，在32位操作系统下最多使用3GB内存。

**内存淘汰策略**

**------“Redis内存满了怎么办？”**

既然可以设置Redis最大占用内存大小，那么配置的内存就有用完的时候。那在内存用完的时候，还继续往Redis里面添加数据不就没内存可用了吗？实际上Redis定义了六种策略用来处理这种情况：

**noeviction(默认策略)**：对于写请求不再提供服务，直接返回错误（DEL请求和部分特殊请求除外）；

**allkeys-lru**：从所有key中使用LRU算法进行淘汰；

**volatile-lru**：从设置了过期时间的key中使用LRU算法进行淘汰 **allkeys-random**：从所有key中随机淘汰数据；

**volatile-random**：从设置了过期时间的key中随机淘汰；

**volatile-ttl**：在设置了过期时间的key中，根据key的过期时间进行淘汰，越早过期的越优先被淘汰。

当使用**volatile-lru、volatile-random、volatile-ttl**这三种策略时，如果没有key可以被淘汰，则和noeviction一样返回错误。

**获取及设置内存淘汰策略**

获取当前内存淘汰策略：

127.0.0.1:6379> config get maxmemory-policy

通过配置文件设置淘汰策略（修改redis.conf文件）：

maxmemory-policy allkeys-lru

通过命令修改淘汰策略：

127.0.0.1:6379> config set maxmemory-policy allkeys-lru

**LRU算法**

**LRU(Least Recently Used)**，即最近最少使用，是一种缓存置换算法。在使用内存作为缓存的时候，缓存的大小一般是固定的。当缓存被占满，这个时候继续往缓存里面添加数据，就需要淘汰一部分老的数据，释放内存空间用来存储新的数据。这个时候就可以使用LRU算法了。其核心思想是：如果一个数据在最近一段时间没有被用到，那么将来被使用到的可能性也很小，所以就可以被淘汰掉。

Redis使用的是近似LRU算法，它跟常规的LRU算法还不太一样。近似LRU算法通过随机采样法淘汰数据，每次随机出5（默认）个key，从里面淘汰掉最近最少使用的key。

可以通过maxmemory-samples参数修改采样数量：例如：maxmemory-samples 10 ，maxmenory-samples配置的越大，淘汰的结果越接近于严格的LRU算法。

Redis为了实现近似LRU算法，给每个key增加了一个额外增加了一个24bit的字段，用来存储该key最后一次被访问的时间。

**Redis3.0对近似LRU的优化**

Redis3.0对近似LRU算法进行了一些优化。新算法会维护一个候选池（大小为16），池中的数据根据访问时间进行排序，第一次随机选取的key都会放入池中，随后每次随机选取的key只有在访问时间小于池中最小的时间才会放入池中，直到候选池被放满。当放满后，如果有新的key需要放入，则将池中最后访问时间最大（最近被访问）的移除。

当需要淘汰的时候，则直接从池中选取最近访问时间最小（最久没被访问）的key淘汰掉就行。

**LFU算法**

LFU算法是Redis4.0里面新加的一种淘汰策略。它的全称是**Least Frequently Used**，它的核心思想是根据key的最近被访问的频率进行淘汰，很少被访问的优先被淘汰，被访问的多的则被留下来。LFU算法能更好的表示一个key被访问的热度。假如你使用的是LRU算法，一个key很久没有被访问到，只刚刚是偶尔被访问了一次，那么它就被认为是热点数据，不会被淘汰，而有些key将来是很有可能被访问到的则被淘汰了。如果使用LFU算法则不会出现这种情况，因为使用一次并不会使一个key成为热点数据。

**LFU一共有两种策略**：

* **volatile-lfu**：在设置了过期时间的key中使用LFU算法淘汰key
* **allkeys-lfu**：在所有的key中使用LFU算法淘汰数据

设置使用这两种淘汰策略跟前面讲的一样，不过要注意的一点是这两种策略只能在Redis4.0及以上设置，如果在Redis4.0以下设置会报错。

1. Redis复制
2. **Redis的复制过程**

Step 1：从节点执行 slaveof 命令。

Step 2：从节点只是保存了 slaveof 命令中主节点的信息，并没有立即发起复制。

Step 3：从节点内部的定时任务发现有主节点的信息，开始使用 socket 连接主节点。

Step 4：连接建立成功后，发送 ping 命令，希望得到 pong 命令响应，否则会进行重连。

Step 5：如果主节点设置了权限，那么就需要进行权限验证，如果验证失败，复制终止。

Step 6：权限验证通过后，进行数据同步，这是耗时最长的操作，主节点将把所有的数据全部发送给从节点。

Step 7：当主节点把当前的数据同步给从节点后，便完成了复制的建立流程。接下来，主节点就会持续的把写命令发送给从节点，保证主从数据一致性。

1. **数据间的同步**

上面说的复制过程，其中有一个步骤是“同步数据集”，这个就是现在讲的“数据间的同步”。Redis 同步有 2 个命令：sync 和 psync，前者是 redis 2.8 之前的同步命令，后者是 redis 2.8 为了优化 sync 新设计的命令。

**psync命令需要3个组件支持**

* 主从节点各自复制偏移量
* 主节点复制积压缓冲区
* 主节点运行 ID

**主从节点各自复制偏移量**

* 参与复制的主从节点都会维护自身的复制偏移量。
* 主节点在处理完写入命令后，会把命令的字节长度做累加记录，统计信息在 info replication 中的 masterreploffset 指标中。
* 从节点每秒钟上报自身的的复制偏移量给主节点，因此主节点也会保存从节点的复制偏移量。
* 从节点在接收到主节点发送的命令后，也会累加自身的偏移量，统计信息在 info replication 中。
* 通过对比主从节点的复制偏移量，可以判断主从节点数据是否一致。

**主节点复制积压缓冲区**

* 复制积压缓冲区是一个保存在主节点的一个固定长度的先进先出的队列，默认大小 1MB。
* 这个队列在 slave 连接是创建。这时主节点响应写命令时，不但会把命令发送给从节点，也会写入复制缓冲区。
* 他的作用就是用于部分复制和复制命令丢失的数据补救。通过 info replication 可以看到相关信息。

**主节点运行ID**

* 每个 redis 启动的时候，都会生成一个 40 位的运行 ID。
* 运行 ID 的主要作用是用来识别 Redis 节点。如果使用 ip+port 的方式，那么如果主节点重启修改了 RDB/AOF 数据，从节点再基于偏移量进行复制将是不安全的。所以，当运行 id 变化后，从节点将进行全量复制。也就是说，redis 重启后，默认从节点会进行全量复制。

**如果在重启时不改变运行ID呢？**

* 可以通过 debug reload 命令重新加载 RDB 并保持运行 ID 不变，从而有效的避免不必要的全量复制。
* 缺点是：debug reload 命令会阻塞当前 Redis 节点主线程，因此对于大数据量的主节点或者无法容忍阻塞的节点，需要谨慎使用。一般通过故障转移机制可以解决这个问题。

**psync命令的使用方式**

* 命令格式为  psync{runId}{offset}
* runId：从节点所复制主节点的运行 id
* offset：当前从节点已复制的数据偏移量

**psync执行流程**

从节点发送 psync 命令给主节点，runId 就是目标主节点的 ID，如果没有默认为 -1，offset 是从节点保存的复制偏移量，如果是第一次复制则为 -1.主节点会根据 runid 和 offset 决定返回结果：

* 如果回复 +FULLRESYNC {runId} {offset} ，那么从节点将触发全量复制流程。
* 如果回复 +CONTINUE，从节点将触发部分复制。
* 如果回复 +ERR，说明主节点不支持 2.8 的 psync 命令，将使用 sync 执行全量复制。

1. **全量复制**

全量复制是 Redis 最早支持的复制方式，也是主从第一次建立复制时必须经历的的阶段。触发全量复制的命令是 sync 和 psync。分水岭版本是 2.8，redis 2.8 之前使用 sync 只能执行全量不同，2.8 之后同时支持全量同步和部分同步。

**流程如下：**

Step 1：发送 psync 命令（spync ？-1）；

Step 2：主节点根据命令返回 FULLRESYNC；

Step 3：从节点记录主节点 ID 和 offset；

Step 4：**主节点 bgsave 并保存 RDB 到本地；**

Step 5：**主节点发送 RBD 文件到从节点；**

Step 6：**从节点收到 RDB 文件并加载到内存中；**

Step 7：主节点在从节点接受数据的期间，将新数据保存到“复制客户端缓冲区”，当从节点加载 RDB 完毕，再发送过去。（如果从节点花费时间过长，将导致缓冲区溢出，最后全量同步失败）；

Step 8：**从节点清空数据后加载 RDB 文件，如果 RDB 文件很大，这一步操作仍然耗时，如果此时客户端访问，将导致数据不一致，可以使用配置slave-server-stale-data 关闭 ；**

Step 9：**从节点成功加载完 RBD 后，如果开启了 AOF，会立刻做 bgrewriteaof。**

**注**：加粗的部分是整个全量同步耗时的地方。

* 如过 RDB 文件大于 6GB，并且是千兆网卡，Redis 的默认超时机制（60 秒），会导致全量复制失败。可以通过调大 repl-timeout 参数来解决此问题。
* Redis 虽然支持无盘复制，即直接通过网络发送给从节点，但功能不是很完善，生产环境慎用。

1. **部分复制**

当从节点正在复制主节点时，如果出现网络闪断和其他异常，从节点会让主节点补发丢失的命令数据，主节点只需要将复制缓冲区的数据发送到从节点就能够保证数据的一致性，相比较全量复制，成本小很多。

* 当从节点出现网络中断，超过了 repl-timeout 时间，主节点就会中断复制连接。
* 主节点会将请求的数据写入到“复制积压缓冲区”，默认 1MB。
* 当从节点恢复，重新连接上主节点，从节点会将 offset 和主节点 id 发送到主节点。
* 主节点校验后，如果偏移量的数后的数据在缓冲区中，就发送 cuntinue 响应 —— 表示可以进行部分复制。
* 主节点将缓冲区的数据发送到从节点，保证主从复制进行正常状态。

1. **心跳**

主从节点在建立复制后，他们之间维护着长连接并彼此发送心跳命令。

**心跳的关键机制：**

* 主从都有心跳检测机制，各自模拟成对方的客户端进行通信，通过 client list 命令查看复制相关客户端信息，主节点的连接状态为 flags = M，从节点的连接状态是 flags = S；
* 主节点默认每隔 10 秒对从节点发送 ping 命令，可修改配置 repl-ping-slave-period 控制发送频率；
* 从节点在主线程每隔一秒发送 replconf ack{offset} 命令，给主节点上报自身当前的复制偏移量；
* 主节点收到 replconf 信息后，判断从节点超时时间，如果超过 repl-timeout 60 秒，则判断节点下线。

为了降低主从延迟，一般把 Redis 主从节点部署在相同的机房/同城机房，避免网络延迟带来的网络分区造成的心跳中断等情况。

1. **异步复制**

主节点不但负责数据读写，还负责把写命令同步给从节点，写命令的发送过程是异步完成，也就是说主节点处理完写命令后立即返回客户度，并不等待从节点复制完成。

异步复制的步骤：

Step 1：主节点接受处理命令。

Step 2：主节点处理完后返回响应结果 。

Step 3：对于修改命令，异步发送给从节点，从节点在主线程中执行复制的命令。

RDB 数据之间的同步非常耗时。所以，Redis 在 2.8 版本退出了类似增量复制的 psync 命令，当 Redis 主从直接发生了网络中断，不会进行全量复制，而是将数据放到缓冲区（默认 1MB）里，再通过主从之间各自维护复制 offset 来判断缓存区的数据是否溢出。如果没有溢出，只需要发送缓冲区数据即可，成本很小；反之，则要进行全量复制。因此控制缓冲区大小非常的重要。

1. 分布式锁

**Redis 分布式锁的作用**

在单机环境下，有个秒杀商品的活动，在短时间内，服务器压力和流量会陡然上升。这个就会存在并发的问题。想要解决并发需要解决一下问题

1. 提高系统吞吐率也就是qps 每秒处理的请求书

解决问题一：采用内存型数据库提高系统的qps；

解决问题二：就要用到经常会遇到的锁，例如MySQL 有读锁、写锁、排他锁、悲观锁、乐观锁。不过这里只讨论redis来实现锁。

**简单版设置锁**

$redis = new Redis();

$redis->connect('127.0.0.1', 6379); //连接Redis

$expire = 10;//有效期10秒

$key = 'lock';//key

$value = time() + $expire;//锁的值 = Unix时间戳 + 锁的有效期

$lock = $redis->setnx($key, $value);

//判断是否上锁成功，成功则执行下步操作

if(!empty($lock))

{

//下步操作...

}

如果以这样的简单版设置锁就能解决所有问题，未免也太小看 锁 在程序中应用了。按正常的操作示例基本上都是这样写的。但是这样写有一些问题：

假如有10000 个请求访问了redis 不存在的键，这样请求就是指接到了MySQL数据，造成CPU短时间内达到100%甚至宕机。这样场景俗称缓存击穿造成的缓存雪崩。

解决问题：引用reids setnx 方法的作用是，当设置的key 不存在时，设置新的值。这样就避免了缓存击穿的问题。检测键的过期时间，避免产生死锁。

**解决死锁问题**

$$expire = 10;//有效期10秒

$key = 'lock';//key

$value = time() + $expire;//锁的值 = Unix时间戳 + 锁的有效期

$status = true;

while($status)

{

$lock = $redis->setnx($key, $value);

if(empty($lock))

{

$value = $redis->get($key);

if($value < time())

{

$redis->del($key);

}

}else{

$status = false;

//下步操作....

}

}

1. 分布式集群业务业务场景下，每台服务器是独立存在的。多台服务器怎么通过一个标识来相互竞争锁呢。这里就用到了分布式锁。以MYSQL 的事务机制来延生。事务四个特性ACID，有四种隔离级别：为提交读、已提交读、可重复读、串行化。这些特性都只在单台服务器上生效。到了分布式集群了，数据在不同的服务器上，紧靠事务很难保持数据的一致性及隔离性，事务的作用就意义不大了。Redis也是如此。

**分布式锁**

//实现Redis分布锁

$key = 'demo'; //要更新信息的缓存KEY

$lockKey = 'lock:'.$key; //设置锁KEY

$lockExpire = 10; //设置锁的有效期为10秒

//获取缓存信息

$result = $redis->get($key);

//判断缓存中是否有数据

if(empty($result))

{

$status = TRUE;

while ($status)

{

//设置锁值为当前时间戳 + 有效期

$lockValue = time() + $lockExpire;

/\*\*

\* 创建锁

\* 试图以$lockKey为key创建一个缓存,value值为当前时间戳

\* 由于setnx()函数只有在不存在当前key的缓存时才会创建成功

\* 所以，用此函数就可以判断当前执行的操作是否已经有其他进程在执行了

\* @var [type]

\*/

$lock = $redis->setnx($lockKey, $lockValue);

/\*\*

\* 满足两个条件中的一个即可进行操作

\* 1、上面一步创建锁成功;

\* 2、 1）判断锁的值（时间戳）是否小于当前时间 $redis->get()

\* 2）同时给锁设置新值成功 $redis->getset()

\*/

if(!empty($lock) || ($redis->get($lockKey) < time() && $redis->getSet($lockKey, $lockValue) < time() ))

{

//给锁设置生存时间

$redis->expire($lockKey, $lockExpire);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//此处执行插入、更新缓存操作...

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//以上程序走完删除锁

//检测锁是否过期，过期锁没必要删除

if($redis->ttl($lockKey))

$redis->del($lockKey);

$status = FALSE;

}else{

/\*\*

\* 如果存在有效锁这里做相应处理

\* 等待当前操作完成再执行此次请求

\* 直接返回

\*/

sleep(2);//等待2秒后再尝试执行操作

}

}

}

1. 面试必问

**1、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：

（1）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。

（2）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。

（3）Redis 支持数据的备份，即 master-slave 模式的数据备份。

**Redis 优势**

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

（2）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。

（3）原子 – Redis 的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过 MULTI 和 EXEC指令包起来。

（4）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？

（1）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。

（2）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。

**2、Redis 的数据类型？**

Redis 支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及 zsetsorted set：有序集合)。

**3、使用 Redis 有哪些好处？**

（1）速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O1)

（2）支持丰富数据类型，支持 string，list，set，Zset，hash 等

（3）支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行

（4）丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除

**4、Redis 相比 Memcached 有哪些优势？**

（1）Memcached 所有的值均是简单的字符串，redis 作为其替代者，支持更为丰富的数据类

（2）Redis 的速度比 Memcached 快；

（3）Redis 可以持久化其数据；

**5、Memcache 与 Redis 的区别都有哪些？**

（1）存储方式 Memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。 Redis 有部份存在硬盘上，这样能保证数据的持久性。

（2）数据支持类型 Memcache 对数据类型支持相对简单。 Redis 有复杂的数据类型。

（3）使用底层模型不同 它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。 Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

**6、Redis 是单进程单线程的？**

Redis 是单进程单线程的，redis 利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销。

**7、一个字符串类型的值能存储最大容量是多少？**

**512M**

**8、Redis 的持久化机制是什么？各自的优缺点？**

Redis提供两种持久化机制 RDB 和 AOF 机制:

1、RDB (Redis DataBase)持久化方式：是指用数据集快照的方式半持久化模式)记录 redis 数据库的所有键值对,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**：

（1）只有一个文件 dump.rdb，方便持久化。

（2）容灾性好，一个文件可以保存到安全的磁盘。

（3）性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以是 IO最大化。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)

（4）相对于数据集大时，比 AOF 的启动效率更高。

**缺点：**

数据安全性低。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。

2、AOF(Append-only file)持久化方式：是指所有的命令行记录以 redis 命令请求协议的格式完全持久化存储)保存为 aof 文件。

**优点：**

（1）数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次命令操作就记录到 aof 文件中一次。

（2）通过 append 模式写文件，即使中途服务器宕机，可以通过 redis-check-aof工具解决数据一致性问题。

（3）AOF 机制的 rewrite 模式。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：**

（1）AOF 文件比 RDB 文件大，且恢复速度慢。

（2）数据集大的时候，比 rdb 启动效率低。

**9、Redis 常见性能问题和解决方案：**

（1）Master 最好不要写内存快照，如果 Master 写内存快照，save 命令调度 rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务；

（2）如果数据比较重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步一；

（3）为了主从复制的速度和连接的稳定性，Master 和 Slave 最好在同一个局域网；

（4）尽量避免在压力很大的主库上增加从；

（5）主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1<- Slave2 <- Slave3…这样的结构方便解决单点故障问题，实现 Slave 对 Master的替换。如果 Master 挂了，可以立刻启用 Slave1 做 Master，其他不变。

**10、redis 过期键的删除策略？**

（1）定时删除:在设置键的过期时间的同时，创建一个定时器 timer). 让定时器在键的过期时间来临时，立即执行对键的删除操作。

（2）惰性删除:放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是否过期，如果过期的话，就删除该键;如果没有过期，就返回该键。

（3）定期删除:每隔一段时间程序就对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。

**11、Redis 的回收策略（淘汰策略）?**

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰；

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰；

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰；

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰；

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰；

no-enviction（驱逐）：禁止驱逐数据；

注意这里的 6 种机制，volatile 和 allkeys 规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的 lru、ttl 以及 random 是三种不同的淘汰策略，再加上一种 no-enviction 永不回收的策略。

**使用策略规则：**

（1）如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用 allkeys-lru；

（2）如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用allkeys-random；

**12、为什么 Redis 需要把所有数据放到内存中？**

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。在内存越来越便宜的今天，redis 将会越来越受欢迎。如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

**13、Redis 的同步机制？**

Redis 可以使用主从同步，从从同步。第一次同步时，主节点做一次 bgsave，并同时将后续修改操作记录到内存 buffer，待完成后将 rdb 文件全量同步到复制节点，复制节点接受完成后将 rdb 镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

**14、Pipeline 有什么好处，为什么要用 pipeline？**

可以将多次 IO 往返的时间缩减为一次，前提是 pipeline 执行的指令之间没有因果相关性。使用 redis-benchmark 进行压测的时候可以发现影响 redis 的 QPS峰值的一个重要因素是 pipeline 批次指令的数目。

**15、Redis 集群，集群的原理是什么？**

（1）Redis Sentinal 着眼于高可用，在 master 宕机时会自动将 slave 提升为master，继续提供服务。

（2）Redis Cluster 着眼于扩展性，在单个 redis 内存不足时，使用 Cluster 进行分片存储。

**16、Redis 集群方案什么情况下会导致整个集群不可用？**

有 A，B，C 三个节点的集群,在没有复制模型的情况下,如果节点 B 失败了，那么整个集群就会以为缺少 5501-11000 这个范围的槽而不可用。

**17、Redis 如何设置密码及验证密码？**

设置密码：config set requirepass 123456

授权密码：auth 123456

**18、 Redis 哈希槽的概念？**

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念，Redis 集群有16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽。

**19、Redis 集群的主从复制模型是怎样的？**

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有 N-1 个复制品.

**20、Redis 集群会有写操作丢失吗？为什么？**

答 ：Redis 并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

**21、Redis 集群之间是如何复制的？**

**异步复制**

**22、Redis 集群最大节点个数是多少？**

**16384 个**

**23、Redis 集群如何选择数据库？**

Redis 集群目前无法做数据库选择，默认在 0 数据库。

**24、怎么测试 Redis 的连通性？**

使用 ping 命令。

**25、怎么理解 Redis 事务？**

（1）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。

（2）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。

**26、Redis 事务相关的命令有哪几个？**

MULTI、EXEC、DISCARD、WATCH

**27、Redis key 的过期时间和永久有效分别怎么设置？**

EXPIRE 和 PERSIST 命令。

**28、Redis 如何做内存优化？**

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的 web 系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的 key,而是应该把这个用户的所有信息存储到一张散列表里面。

**29、Redis 回收进程如何工作的？**

一个客户端运行了新的命令，添加了新的数据。Redi 检查内存使用情况，如果大于 maxmemory 的限制, 则根据设定好的策略进行回收。一个新的命令被执行，等等。所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地回收回到边界以下。如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越。

**30、都有哪些办法可以降低 Redis 的内存使用情况呢？**

如果使用的是 32 位的 Redis 实例，可以好好利用 Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的 Key-Value 可以用更紧凑的方式存放到一起。

**31、Redis 的内存用完了会发生什么？**

如果达到设置的上限，Redis 的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以将 Redis 当缓存来使用配置淘汰机制，当 Redis 达到内存上限时会冲刷掉旧的内容。

**32、一个 Redis 实例最多能存放多少的 keys？List、Set、Sorted Set 他们最多能存放多少元素？**

理论上 Redis 可以处理多达 232 的 keys，并且在实际中进行了测试，每个实例至少存放了 2 亿 5 千万的 keys。我们正在测试一些较大的值。任何 list、set、和 sorted set 都可以放 232 个元素。换句话说，Redis 的存储极限是系统中的可用内存值。

**33、MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据？**

Redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。Redis 提供 6 种数据淘汰策略：

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据

**34、Redis 最适合的场景？**

1、会话缓存（Session Cache）

最常用的一种使用 Redis 的情景是会话缓存（session cache）。用 Redis 缓存会话比其他存储（如 Memcached）的优势在于：Redis 提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？ 幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用 Redis 来缓存会话的文档。甚至广为人知的商业平台Magento 也提供 Redis 的插件。

2、全页缓存（FPC）

除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问题，即使重启了 Redis 实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。 再次以 Magento 为例，Magento提供一个插件来使用 Redis 作为全页缓存后端。 此外，对 WordPress 的用户来说，Pantheon 有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

3、队列

Reids 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis能作为一个很好的消息队列平台来使用。Redis 作为队列使用的操作，就类似于本地程序语言（如 Python）对 list 的 push/pop 操作。 如果你快速的在 Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用 Redis 创建非常好的后端工具，以满足各种队列需求。例如，Celery 有一个后台就是使用 Redis 作为 broker，你可以从这里去查看。

4，排行榜/计数器

Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的 10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可： 当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行： ZRANGE user\_scores 0 10 WITHSCORES Agora Games 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。

5、发布/订阅

发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建立聊天系统。

**35、Reids的特点**

Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。

Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存多种数据结构，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性 能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一 个功能加强版的memcached来用。

　Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

**36、redis常见性能问题和解决方案：**

1). Master写内存快照，save命令调度rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以Master最好不要写内存快照。

2). Master AOF持久化，如果不重写AOF文件，这个持久化方式对性能的影响是最小的，但是AOF文件会不断增大，AOF文件过大会影响Master重启的恢复速度。Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化,如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。

3). Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。

4). Redis主从复制的性能问题，为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内。

**37、redis的并发竞争问题如何解决?**

　Redis为单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，但是在Jedis客户端对Redis进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是

　由于客户端连接混乱造成。对此有2种解决方法：

1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。

2.服务器角度，利用setnx实现锁。

注：对于第一种，需要应用程序自己处理资源的同步，可以使用的方法比较通俗，可以使用synchronized也可以使用lock；第二种需要用到Redis的setnx命令，但是需要注意一些问题。

**38、redis事物的了解CAS(check-and-set 操作实现乐观锁)?**

和众多其它数据库一样，Redis作为NoSQL数据库也同样提供了事务机制。在Redis中，MULTI/EXEC/DISCARD/WATCH这四个命令是我们实现事务的基石。Redis中事务的实现特征：

1). 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

2). 和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。

3). 通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行EXEC/DISCARD命令来提交/回滚该事务内的所有操作。这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。

4). 在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行。

5). 当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。

Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

**39、WATCH命令和基于CAS的乐观锁：**

在Redis的事务中，WATCH命令可用于提供CAS(check-and-set)功能。假设我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，EXEC命令执行的事务都将被放弃，同时返回Null multi-bulk应答以通知调用者事务

　执行失败。例如再次假设Redis中并未提供incr命令来完成键值的原子性递增，如果要实现该功能，我们只能自行编写相应的代码。其伪码如下：

　　val = GET mykey

　　val = val + 1

　　SET mykey $val

以上代码只有在单连接的情况下才可以保证执行结果是正确的，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的一种错误场景--竞态争用(race condition)。比如，客户端A和B都在同一时刻读取了mykey的原有值，假设该值为10，此后两个客户端又均将该值加一后set回Redis服务器，这样就会导致mykey的结果为11，而不是12。为了解决类似的问题，需要借助WATCH命令的帮助，见如下代码：

　　WATCH mykey

　　val = GET mykey

　　val = val + 1

　　MULTI

　　SET mykey $val

　　EXEC

和此前代码不同的是，新代码在获取mykey的值之前先通过WATCH命令监控了该键，此后又将set命令包围在事务中，这样就可以有效的保证每个连接在执行EXEC之前，如果当前连接获取的mykey的值被其它连接的客户端修改，那么当前连接的EXEC命令将执行失败。这样调用者在判断返回值后就可以获悉val是否被重新设置成功。

**40、redis持久化的几种方式**

1、快照（snapshots）

缺省情况情况下，Redis把数据快照存放在磁盘上的二进制文件中，文件名为dump.rdb。你可以配置Redis的持久化策略，例如数据集中每N秒钟有超过M次更新，就将数据写入磁盘；或者你可以手工调用命令SAVE或BGSAVE。

　　工作原理

　　． Redis forks.

　　． 子进程开始将数据写到临时RDB文件中。

　　． 当子进程完成写RDB文件，用新文件替换老文件。

　　． 这种方式可以使Redis使用copy-on-write技术。

2、AOF

快照模式并不十分健壮，当系统停止，或者无意中Redis被kill掉，最后写入Redis的数据就会丢失。这对某些应用也许不是大问题，但对于要求高可靠性的应用来说，

　　Redis就不是一个合适的选择。

　　Append-only文件模式是另一种选择。

　　你可以在配置文件中打开AOF模式

3、虚拟内存方式

当key很小而value很大时,使用VM的效果会比较好.因为这样节约的内存比较大.

当你的key不小时,可以考虑使用一些非常方法将很大的key变成很大的value,比如你可以考虑将key,value组合成一个新的value.

vm-max-threads这个参数,可以设置访问swap文件的线程数,设置最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的.可能会造成比较长时间的延迟,但是对数据完整性有很好的保证.

**41、redis的缓存失效策略和主键失效机制**

作为缓存系统都要定期清理无效数据，就需要一个主键失效和淘汰策略.在Redis当中，有生存期的key被称为volatile。在创建缓存时，要为给定的key设置生存期，当key过期的时候（生存期为0），它可能会被删除。

　　1、影响生存时间的一些操作

生存时间可以通过使用 DEL 命令来删除整个 key 来移除，或者被 SET 和 GETSET 命令覆盖原来的数据，也就是说，修改key对应的value和使用另外相同的key和value来覆盖以后，当前数据的生存时间不同。

比如说，对一个 key 执行INCR命令，对一个列表进行LPUSH命令，或者对一个哈希表执行HSET命令，这类操作都不会修改 key 本身的生存时间。另一方面，如果使用RENAME对一个 key 进行改名，那么改名后的 key的生存时间和改名前一样。

RENAME命令的另一种可能是，尝试将一个带生存时间的 key 改名成另一个带生存时间的 another\_key ，这时旧的 another\_key (以及它的生存时间)会被删除，然后旧的 key 会改名为 another\_key ，因此，新的 another\_key 的生存时间也和原本的 key 一样。使用PERSIST命令可以在不删除 key 的情况下，移除 key 的生存时间，让 key 重新成为一个persistent key 。

　　2、如何更新生存时间

可以对一个已经带有生存时间的 key 执行EXPIRE命令，新指定的生存时间会取代旧的生存时间。过期时间的精度已经被控制在1ms之内，主键失效的时间复杂度是O（1），

EXPIRE和TTL命令搭配使用，TTL可以查看key的当前生存时间。设置成功返回 1；当 key 不存在或者不能为 key 设置生存时间时，返回 0 。

　　最大缓存配置

　　在 redis 中，允许用户设置最大使用内存大小

　　server.maxmemory

　　默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。

**redis 提供 6种数据淘汰策略：**

* volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰；
* volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰；
* volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰；
* allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰；
* allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰；
* no-enviction（驱逐）：禁止驱逐数据；

注意6种机制，volatile和allkeys规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的lru、ttl以及random是三种不同的淘汰策略，再加上一种no-enviction永不回收的策略。

**使用策略规则：**

　1、如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用allkeys-lru；

　2、如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用allkeys-random。

**三种数据淘汰策略：**

ttl和random比较容易理解，实现也会比较简单。主要是Lru最近最少使用淘汰策略，设计上会对key 按失效时间排序，然后取最先失效的key进行淘汰。

1. MongoDB
2. Memcache
3. Web安全及防范策略

浏览器安全其实是受**同源策略（同域名、同端口、同协议名**）保护的，但是<script>、<img>、<iframe>、<link>等标签都可以跨域加载资源，而不受同源策略的限制。

这些带"src"属性的标签每次加载时，浏览器会发起一次GET请求。通过src属性加载的资源，浏览器限制了javascript的权限，使其不能读、写返回的内容。  
常见的六大安全问题有：**XSS、CSRF、点击劫持、URL跳转漏洞、SQL注入、OS命令注入攻击。**

1. **XSS 跨站脚本攻击**

XSS (Cross-Site Scripting)，跨站脚本攻击，因为缩写和 CSS重叠，所以只能叫 XSS。XSS 的**原理是**恶意攻击者往 Web 页面里插入恶意可执行网页脚本代码，当用户浏览该页之时，嵌入其中 Web 里面的脚本代码会被执行，从而可以达到攻击者盗取用户信息或其他侵犯用户安全隐私的目的。

**非持久型XSS（反射型XSS）**一般是通过给别人发送带有恶意脚本代码参数的 URL，当 URL 地址被打开时，特有的恶意代码参数被 HTML 解析、执行。

**特点：**

1. 即时性，不经过服务器存储，直接通过 HTTP 的 GET 和 POST 请求就能完成一次攻击，拿到用户隐私数据。  
   2 攻击者需要诱骗点击,必须要通过用户点击链接才能发起  
   3 反馈率低，所以较难发现和响应修复  
   4 盗取用户敏感保密信息

**持久型XSS（存储型XSS）**持久型 XSS 漏洞，一般存在于 **Form 表单提交**等交互功能，如文章留言，提交文本信息等，黑客利用的 XSS 漏洞，将内容经正常功能提交进入**数据库持久保存**，当前端页面获得后端从数据库中读出的注入代码时，恰好将其渲染执行。

**特点：**

1 持久性，植入在数据库中  
2. 盗取用户敏感私密信息  
3. 危害面广

**防范策略：**

1 设置白名单CSP。通常可以通过两种方式来开启 CSP：  
设置 HTTP Header 中的 Content-Security-Policy；  
设置 meta 标签的方式 <meta http-equiv="Content-Security-Policy">；  
Content-Security-Policy: default-src 'self' //只允许加载本站资源  
Content-Security-Policy: img-src https://\* //只允许加载 HTTPS 协议图片  
Content-Security-Policy: child-src 'none' //允许加载任何来源框架

2 用户的输入永远不可信任的，最普遍的做法就是转义输入输出的内容，对于引号、尖括号、斜杠进行转义

3 HttpOnly Cookie 这是预防XSS攻击窃取用户cookie最有效的防御手段。Web应用程序在设置cookie时，将其属性设为HttpOnly，就可以避免该网页的cookie被客户端恶意JavaScript窃取，保护用户cookie信息。

1. **CSRF 跨站请求伪造**

CSRF(Cross Site Request Forgery)，即跨站请求伪造，是一种常见的Web攻击，它利用用户已登录的身份，在用户毫不知情的情况下，以用户的名义完成非法操作。

**防范策略：**

1 可以对 Cookie 设置 **SameSite 属性**。该属性表示 Cookie 不随着跨域请求发送，可以很大程度减少 CSRF 的攻击，但是该属性目前并不是所有浏览器都兼容；

2 通过Referer限制！Referer信息告诉服务器是从**哪个页面链接**过来的；

3 目前比较完善的解决方案是加入**Anti-CSRF-Token**,即发送请求时在HTTP 请求中以参数的形式加入一个随机产生的token，服务器读取浏览器当前域cookie中这个token值，进行比较校验。

4 **验证码操作**，体验上不是很好。

1. **点击劫持**

通过iframe透明化，漏出来一个按钮，诱惑用户点击。

**防范策略：**

1 X-FRAME-OPTIONS是一个 HTTP 响应头，在现代浏览器有一个很好的支持。这个 HTTP 响应头 就是为了防御用 iframe 嵌套的点击劫持攻击。X-FRAME-OPTIONS有3个值可选：  
DENY，表示页面不允许通过 iframe 的方式展示  
SAMEORIGIN，表示页面可以在相同域名下通过 iframe 的方式展示  
ALLOW-FROM，表示页面可以在指定来源的 iframe 中展示  
 2 通过js的方法判断，来隐藏iframe显示的页面。

1. **URL跳转漏洞**

借助未验证的URL跳转，将应用程序引导到不安全的第三方区域，从而导致的安全问题。

**防范策略：**

1 referer的限制 如果确定传递URL参数进入的来源，我们可以通过该方式实现安全限制，保证该URL的有效性，避免恶意用户自己生成跳转链接  
 2 加入有效性验证Token 保证所有生成的链接都是来自于我们可信域的，通过在生成的链接里加入用户不可控的Token对生成的链接进行校验，可以避免用户生成自己的恶意链接从而被利用，但是如果功能本身要求比较开放，可能导致有一定的限制。

1. **SQL注入**

SQL注入的本质:数据和代码未分离，即数据当做了代码来执行。  
有可能会造成获取数据库信息、管理员后台用户名和密码、读取服务器敏感文件等问题，甚至修改数据库内容。

**防范策略：**

1 严格限制Web应用的数据库的操作权限，给此用户提供仅仅能够满足其工作的最低权限，从而最大限度的减少注入攻击对数据库的危害  
 2 后端代码检查输入的数据是否符合预期，严格限制变量的类型，例如使用正则表达式进行一些匹配处理。  
 3 对进入数据库的特殊字符（'，"，，<，>，&，，; 等）进行转义处理，或编码转换\*。基本上所有的后端语言都有对字符串进行转义处理的方法，比如 lodash 的 lodash.\_escapehtmlchar 库。  
 4 所有的查询语句建议使用数据库提供的参数化查询接口，参数化的语句使用参数而不是将用户输入变量嵌入到 SQL 语句中，即不要直接拼接 SQL 语句。

1. **OS命令注入**

OS命令注入攻击指通过Web应用，执行非法的操作系统命令达到攻击的目的。

**防范策略：**

1 后端对前端提交内容进行规则限制（比如正则表达式）。  
 2 在调用系统命令前对所有传入参数进行命令行参数转义过滤。  
 3 不要直接拼接命令语句，借助一些工具做拼接、转义预处理，例如 Node.js 的 shell-escape npm包。

1. 缓存方案
2. 日志记录
3. Nginx与负载均衡
4. 秒杀超卖问题
5. 接口规范 加密