**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。

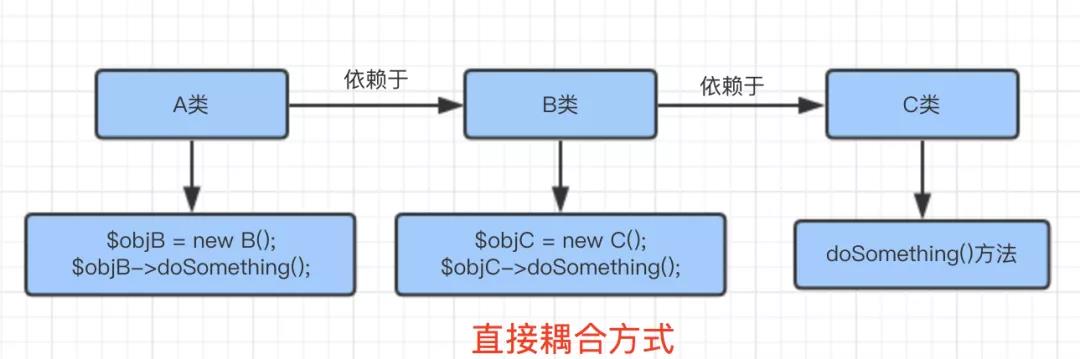
**DI - Dependency Injection 依赖注入**：依赖注入是从应用程序的角度在描述，可以把依赖注入理解为：应用程序依赖容器创建并注入它所需要的外部资源。

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。

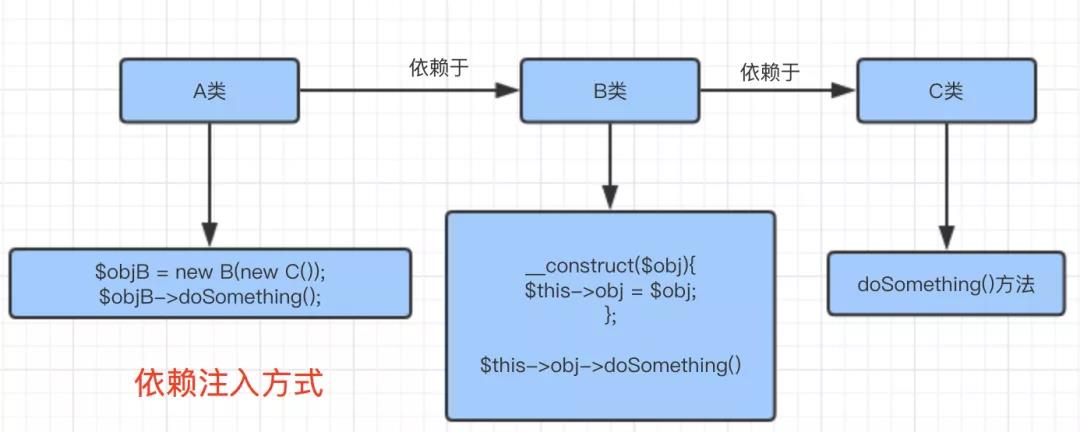
1. 原理

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。



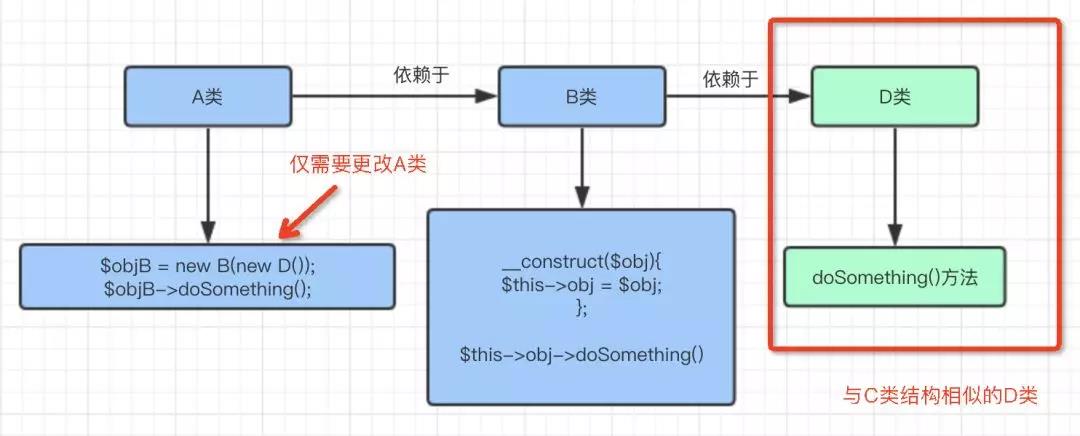
而依赖注入方式如下：



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.');

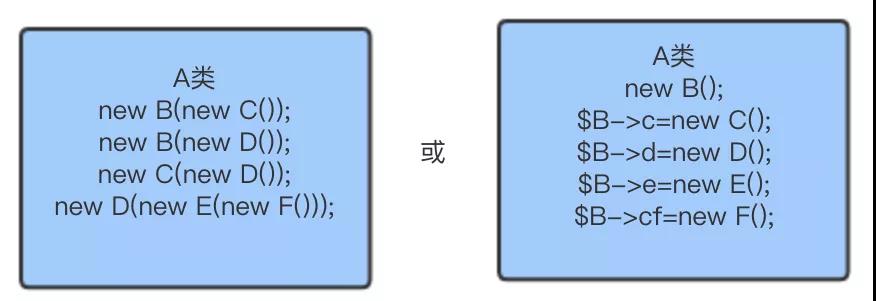
}

}

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。

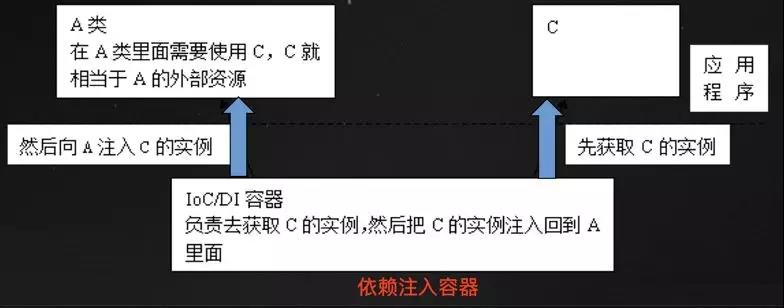
1. 依赖注入容器（IoC容器）

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';

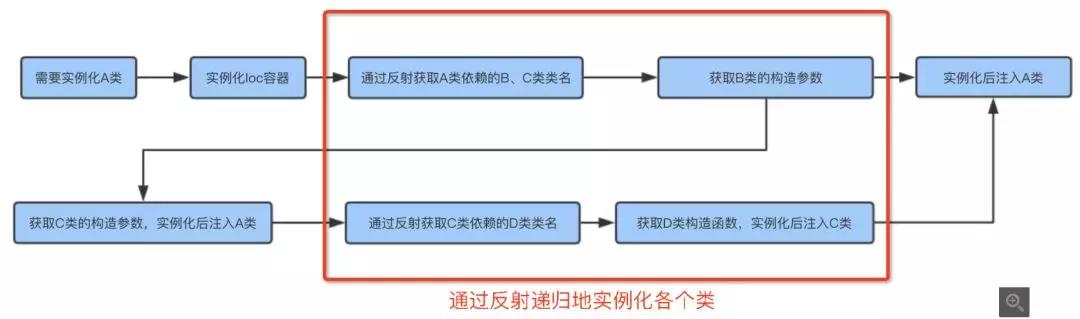
$company = $di->company;

$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
2. 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;

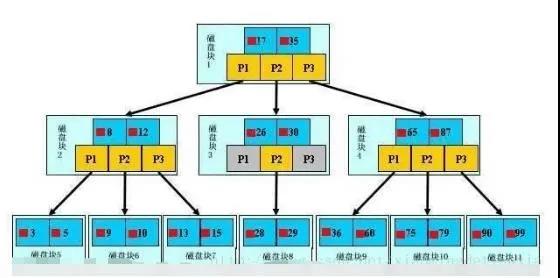
**索引设置的原则**

* 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。
2. 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；

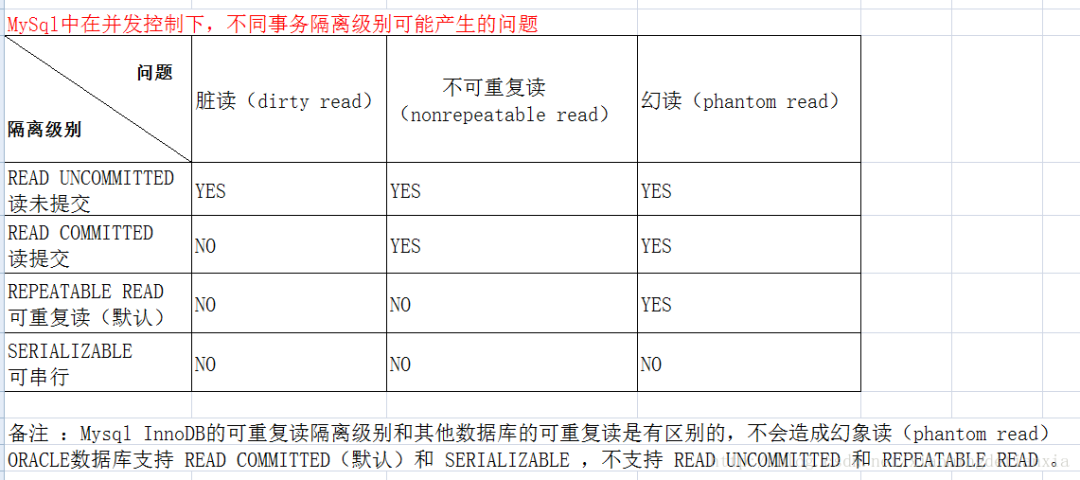
**RELEASE SAVEPOINT identifier**；删除一个事务的保存点，当没有指定的保存点时，执行该语句会抛出一个异常；

**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。

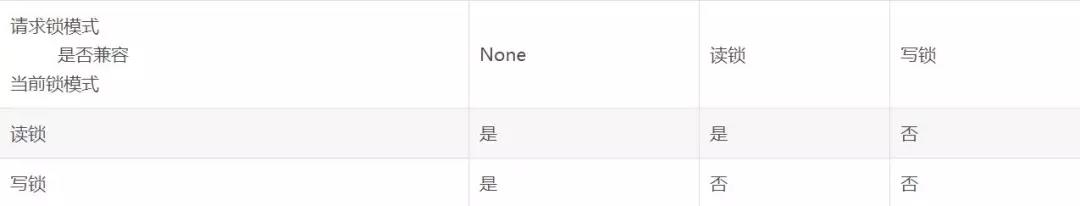
1. 超键 候选键 主键 外键

**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！

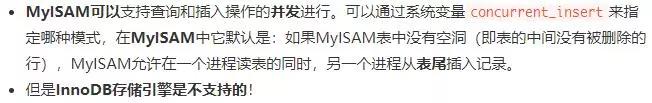


从上面已经看到了：读锁和写锁是互斥的，读写操作是串行。

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. Redis与MongoDB、Memcache
2. Web安全策略
3. 缓存方案
4. Nginx与负载均衡
5. 秒杀超卖问题