安全&性能&架构

点滴学习，随时记录

缓存穿透、缓存击穿、缓存雪崩的理解

缓存处理流程

前台请求，后台先从缓存中取数据，取到直接返回结果，取不到时从数据库中取，数据库取到更新缓存，并返回结果，数据库也没取到，那直接返回空结果。

缓存穿透

描述：

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

解决方案：

接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；

从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

缓存击穿

描述：

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，（若对应的数据库查询又比较慢，会造成更大的压力）造成过大压力。

解决方案：

若数据库操作慢，首选解决慢操作问题

设置热点数据永远不过期。如电商项目中的爆款。

加互斥锁，若缓存中没有数据，第1个进入的线程，获取锁并从数据库去取数据，没释放锁之前，其他并行进入的线程会等待100ms，再重新去缓存取数据。或直接终止请求，前端展示请求失败提示，这样就防止都去数据库重复取数据，重复往缓存中更新数据情况出现。

缓存雪崩

描述：

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

另一种雪崩是缓存服务器某个节点宕机或断网，节点的宕机，对数据库服务器造成的压力是不可预知的，很有可能瞬间就把数据库压垮。

解决方案：

缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。以电商项目为例，一般是采取不同分类商品，缓存不同周期。在同一分类中的商品，加上一个随机因子。这样能尽可能分散缓存过期时间，而且，热门类目的商品缓存时间长一些，冷门类目的商品缓存时间短一些，也能节省缓存服务的资源。

如果缓存数据库是分布式部署，将热点数据均匀分布在不同得缓存数据库中。

设置热点数据永远不过期。

何为跨域

同源策略，它是由Netscape提出的一个著名的安全策略。现在所有支持JavaScript 的浏览器都会使用这个策略。所谓同源是指，域名，协议，端口相同。当一个浏览器的两个tab页中分别打开来 百度和谷歌的页面当一个百度浏览器执行一个脚本的时候会检查这个脚本是属于哪个页面的，即检查是否同源，只有和百度同源的脚本才会被执行。

具体可以查看下表：

URL	说明	是否允许通信
http://www.a.com/a.js http://www.a.com/b.js	同一域名下	允许
http://www.a.com/lab/a.js http://www.a.com/script/b.js	同一域名下不同文件夹	允许
http://www.a.com:8000/a.js http://www.a.com/b.js	同一域名，不同端口	不允许
http://www.a.com/a.js https://www.a.com/b.js	同一域名，不同协议	不允许
http://www.a.com/a.js http://70.32.92.74/b.js	域名和域名对应ip	不允许
http://www.a.com/a.js http://script.a.com/b.js	主域相同，子域不同	不允许
http://www.a.com/a.js http://a.com/b.js	同一域名，不同二级域名（同上）	不允许（cookie这种情况下也不允许访问）
http://www.cnblogs.com/a.js http://www.a.com/b.js	不同域名	不允许

而解决这个问题的方法即为跨域。

浏览器 跨域资源共享 cors 详解

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。

它允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。

本文详细介绍CORS的内部机制。

一、简介

CORS需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE浏览器不能低于IE10。

整个CORS通信过程，都是浏览器自动完成，不需要用户参与。对于开发者来说，CORS通信与同源的AJAX通信没有差别，代码完全一样。浏览器一旦发现AJAX请求跨源，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感觉。

因此，实现CORS通信的关键是服务器。只要服务器实现了CORS接口，就可以跨源通信。

二、两种请求

浏览器将CORS请求分成两类：简单请求（simple request）和非简单请求（not-so-simple request）。

只要同时满足以下两大条件，就属于简单请求。

```
(1) 请求方法是以下三种方法之一：
    HEAD
    GET
    POST

(2) HTTP的头信息不超出以下几种字段：
    Accept
    Accept-Language
    Content-Language
    Last-Event-ID
```

```
11     Content-Type：只限于三个值application/x-www-form-urlencoded、multip
   art/form-data、text/plain
```

凡是不同时满足上面两个条件，就属于非简单请求。

浏览器对这两种请求的处理，是不一样的。

三、简单请求

3.1 基本流程

对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个Origin字段。

下面是一个例子，浏览器发现这次跨源AJAX请求是简单请求，就自动在头信息之中，添加一个Origin字段。

```
1 GET /cors HTTP/1.1
2 Origin: http://api.bob.com
3 Host: api.alice.com
4 Accept-Language: en-US
5 Connection: keep-alive
6 User-Agent: Mozilla/5.0...
```

上面的头信息中，Origin字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口）。服务器根据这个值，决定是否同意这次请求。

如果Origin指定的源，不在许可范围内，服务器会返回一个正常的HTTP回应。浏览器发现，这个回应的头信息没有包含Access-Control-Allow-Origin字段（详见下文），就知道出错了，从而抛出一个错误，被XMLHttpRequest的onerror回调函数捕获。注意，这种错误无法通过状态码识别，因为HTTP回应的状态码有可能是200。

如果Origin指定的域名在许可范围内，服务器返回的响应，会多出几个头信息字段。

```
1 Access-Control-Allow-Origin: http://api.bob.com
2 Access-Control-Allow-Credentials: true
```

```
Access-Control-Expose-Headers: FooBar
Content-Type: text/html; charset=utf-8
```

上面的头信息之中，有三个与CORS请求相关的字段，都以Access-Control-开头。

（1）Access-Control-Allow-Origin

该字段是必须的。它的值要么是请求时Origin字段的值，要么是一个*，表示接受任意域名的请求。

（2）Access-Control-Allow-Credentials

该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。这个值也只能设为true，如果服务器不要浏览器发送Cookie，删除该字段即可。

（3）Access-Control-Expose-Headers

该字段可选。CORS请求时，XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。如果想拿到其他字段，就必须在Access-Control-Expose-Headers里面指定。上面的例子指定，getResponseHeader('FooBar')可以返回FooBar字段的值。

3.2 withCredentials 属性

上面说到，CORS请求默认不发送Cookie和HTTP认证信息。如果要把Cookie发到服务器，一方面要服务器同意，指定Access-Control-Allow-Credentials字段。

```
Access-Control-Allow-Credentials: true
```

另一方面，开发者必须在AJAX请求中打开withCredentials属性。

```
var xhr = new XMLHttpRequest();
xhr.withCredentials = true;
```

否则，即使服务器同意发送Cookie，浏览器也不会发送。或者，服务器要求设置Cookie，浏览器也不会处理。

但是，如果省略withCredentials设置，有的浏览器还是会一起发送Cookie。这时，可以显式关闭withCredentials。

```
xhr.withCredentials = false;
```

需要注意的是，如果要发送Cookie，Access-Control-Allow-Origin就不能设为星号，必须指定明确的、与请求网页一致的域名。同时，Cookie依然遵循同源政策，只有用服务器域名设置的Cookie才会上传，其他域名的Cookie并不会上传，且（跨源）原网页代码中的document.cookie也无法读取服务器域名下的Cookie。

四、非简单请求

4.1 预检请求

非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。

非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight）。

浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的XMLHttpRequest请求，否则就报错。

下面是一段浏览器的JavaScript脚本。

```
var url = 'http://api.alice.com/cors';
var xhr = new XMLHttpRequest();
xhr.open('PUT', url, true);
xhr.setRequestHeader('X-Custom-Header', 'value');
xhr.send();
```

上面代码中，HTTP请求的方法是PUT，并且发送一个自定义头信息X-Custom-Header。

浏览器发现，这是一个非简单请求，就自动发出一个"预检"请求，要求服务器确认可以这样请求。下面是这个"预检"请求的HTTP头信息。

```
OPTIONS /cors HTTP/1.1
Origin: http://api.bob.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-Custom-Header
Host: api.alice.com
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

"预检"请求用的请求方法是OPTIONS，表示这个请求是用来询问的。头信息里面，关键字段是Origin，表示请求来自哪个源。

除了Origin字段，"预检"请求的头信息包括两个特殊字段。

（1）Access-Control-Request-Method

该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法，上例是PUT。

（2）Access-Control-Request-Headers

该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段，上例是X-Custom-Header。

4.2 预检请求的回应

服务器收到"预检"请求以后，检查了Origin、Access-Control-Request-Method和Access-Control-Request-Headers字段以后，确认允许跨源请求，就可以做出回应。

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
```

```
Access-Control-Allow-Origin: http://api.bob.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: X-Custom-Header
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

上面的HTTP回应中，关键的是Access-Control-Allow-Origin字段，表示http://api.bob.com可以请求数据。该字段也可以设为星号，表示同意任意跨源请求。

```
Access-Control-Allow-Origin: *
```

如果浏览器否定了"预检"请求，会返回一个正常的HTTP回应，但是没有任何CORS相关的头信息字段。这时，浏览器就会认定，服务器不同意预检请求，因此触发一个错误，被XMLHttpRequest对象的onerror回调函数捕获。控制台会打印出如下的报错信息。

```
XMLHttpRequest cannot load http://api.alice.com.
Origin http://api.bob.com is not allowed by Access-Control-Allow-Orig
in.
```

服务器回应的其他CORS相关字段如下。

```
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: X-Custom-Header
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 1728000
```

（1）Access-Control-Allow-Methods

该字段必需，它的值是逗号分隔的一个字符串，表明服务器支持的所有跨域请求的方法。注意，返回的是所有支持的方法，而不单是浏览器请求的那个方法。这是为了避免多次"预检"请求。

（2）Access-Control-Allow-Headers

如果浏览器请求包括Access-Control-Request-Headers字段，则Access-Control-Allow-Headers字段是必需的。它也是一个逗号分隔的字符串，表明服务器支持的所有头信息字段，不限于浏览器在"预检"中请求的字段。

（3）Access-Control-Allow-Credentials

该字段与简单请求时的含义相同。

（4）Access-Control-Max-Age

该字段可选，用来指定本次预检请求的有效期，单位为秒。上面结果中，有效期是20天（1728000秒），即允许缓存该条回应1728000秒（即20天），在此期间，不用发出另一条预检请求。

4.3 浏览器的正常请求和回应

一旦服务器通过了"预检"请求，以后每次浏览器正常的CORS请求，就都跟简单请求一样，会有一个Origin头信息字段。服务器的回应，也都会有一个Access-Control-Allow-Origin头信息字段。

下面是"预检"请求之后，浏览器的正常CORS请求。

```
PUT /cors HTTP/1.1
Origin: http://api.bob.com
Host: api.alice.com
X-Custom-Header: value
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

上面头信息的Origin字段是浏览器自动添加的。

下面是服务器正常的回应。

```
Access-Control-Allow-Origin: http://api.bob.com
Content-Type: text/html; charset=utf-8
```

上面头信息中，Access-Control-Allow-Origin字段是每次回应都必定包含的。

五、与JSONP的比较

CORS与JSONP的使用目的相同，但是比JSONP更强大。

JSONP只支持GET请求，CORS支持所有类型的HTTP请求。JSONP的优势在于支持老式浏览器，以及可以向不支持CORS的网站请求数据。

转自：阮一峰的网络日志

参考文章：

Http访问控制（cors）

跨域方法汇总

javascript get(jsonp)方式跨域：

原理：<script>标签的src属性并不被同源策略所约束，所以可以获取任何服务器上脚本并执行。
html页面添加以下js代码：

```
    # result方法和接口callback参数值一致，用来获取服务端返回数据，
    function result(data) {
        //此处弹出返回值的message,若不需要返回值，result方法可不写
        alert(data.message);
    }
    //添加<script>标签的方法
    function addScriptTag(src){
            var script = document.createElement('script');
        script.setAttribute("type","text/javascript");
        script.src = src;
        document.body.appendChild(script);
    }

    window.onload = function(){
        addScriptTag("http://localhost/test.php?v=1.0&q=apple&callba
ck=result");
    }
```

被请求的php文件，返回json字符串：

```
<?php
//获取回调函数名
$result = $_POST['callback'];
//返回json数据
$json_data  = '{"status":1,"message":"success"}';
echo $result.'('.$json_data .')';
?>
```

javascript post 方式跨域：

post方法跨域有两种方案：
一是，建立同源代理，通过代理发送请求获取返回值
二是，注入js脚本，插入iframe，在iframe中插入from，通过from提交数据，发送请求

```
function funPostBack(srvMethod){
        var contentNR=$(document.getElementById("reportFrame").contentWindow.document).find("#content-container div.pageContentDIV.contentDIV").html();
        if(document.getElementById("SMAL")!=null)
        {
            document.getElementById("SMAL").remove();//首先删除
        }
        var form ="<form action='"+postbackUrl+"' method='post'>" +
                "<input type='hidden' name='parms' value=''/> " +
                "<input type='hidden' name='srvMethod' value=''/> " +
                "</form> ";
        $("body").append("<iframe id='SMAL' name='SMAL' style='display: none'></iframe>");//载入iframe
        $( "#SMAL" ).contents().find('body').html(form);//将form表单塞入iframe;
        $( "#SMAL" ).contents().find("form input[name='parms']").val(contentNR);
        $( "#SMAL" ).contents().find("form input[name='method']").val(srvMethod);
        $( "#SMAL" ).contents().find('form').submit();
}
```

注意：post方法跨域最好不要用

jquery ajax get 方法跨域：

ajax法：

```
$.ajax({
    //此处?表示person
        url:"http://visit_log/target_log.php?callback={result}&category=vale1&name=vale2&action=value3&value=value4",
        dataType:"jsonp",
        jsonpCallback:"person",
        success:function(data){
            alert(data.status + " is a a" + data.message);
        }
});
```

被请求的php文件，返回json字符串：

```
<?php
//获取回调函数名
$result = $_POST['callback'];
//返回json数据
$json_data  = '{"status":1,"message":"success"}';
echo result.'('.$json_data .')';
?>
```

getjson法：

```
<script>
$.getJSON("http://visit_log/target_log.php?callback={result}&category=vale1&name=vale2&action=value3&value=value4",function(data){
        alert(data.status + " is a a" + data.message);
});
</script>
```

iframe 跨域

window.name 跨域：

原理：

name 在浏览器环境中是一个全局window对象的属性，在一个窗口中赋了window.name之后，无论怎么刷新该窗口其window.name属性都不会变。但window.name 属性仅对相同域名的 iframe 可访问。

因此我们可以在异域目标页面将要获取的数据赋值给window.name ,此时对该窗口进行换域，换成本域url，此时本域即可访问赋值的window.name值

实例：

```
//异域页面赋值：www.ueditor.com
window.name='';
function select_list(modelid, id) {
    var modeid = modelid + 'n' + id;
    var result = window.name;
        if(result =='' ) {
            window.name = modeid;
        }
 }
```

```
//本域对异域进行换域并取值
//proxy_url 相当于代理页面，可以为空文件，放在本域任意目录下即可
var proxy_url = 'http://www.baidu.com/wzlist_example/proxy.html';
//换域
iframe.src = proxy_url;
if (iframe.attachEvent){
    iframe.attachEvent("onload", function(){
        //取值
        result = iframe.contentWindow.name;
    });
} else {
    iframe.onload = function(){
        ////取值
        result = iframe.contentWindow.name;
    };
}
```

window.name 的优势：

```
数据量更大（2M）
更安全
可传递多种数据格式
可跨所有域，主域、子域均可跨
window.name 的劣势：
只适用于隐藏iframe的情形（因中间过程需要换域，若换域后iframe仍显示则页面发生了变化，不再是原来页面）
```

document.domain跨域：

document.domain方式只能跨主域相同的页面

主域相同的页面只要 document.domain 相同即可互相访问
a页面：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>A</title>
</head>
<body>
<textarea id="message">这是高层的密码！</textarea><br/>
<button id="test">看看员工在说什么</button><br/><br/><br/>员工们：<br/>
<iframe src="http://b.xxx.com/js/crossdomain/demo/b.htm" width="500" height="300" id="iframe"></iframe>
<script>
   document.domain = "jiaju.com";
   document.getElementByI d("test").onclick = function(){
        alert(document.getElementByI d("iframe").contentWindow.document.getElementByI d("message").value);
    }
</script>
</body>
</html>
```

b页面

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
>
<title>JSONP方式</title>
<script type="text/javascript" src="/js/jquery-1.5.1.min.js"></scrip
t>
</head>
<body>
<textarea id="message">这是员工的密码！</textarea><br/>
<button id="test">看看领导在说什么</button><br/>
<script>
    document.domain = "jiaju.com";
    document.getElementByI d("test").onclick = function(){
        alert(parent.document.getElementByI d("message").value);
    }
</script>
</body>
</html>
```

两个域都设置：document.domain=‘jiaju.com’

php cors 跨域

通过设置Access-Control-Allow-Origin来实现跨域。

例如：客户端的域名是client.runoob.com，而请求的域名是server.runoob.com。
如果直接使用ajax访问，会有以下错误：
XMLHttpRequest cannot load http://server.runoob.com/server.php. No 'Access-Control-Allow-Origin' header is present on the requested resource.Origin 'http://client.runoob.com' is therefore not allowed access.

允许单个域名跨域访问

指定某域名（http://client.runoob.com）跨域访问，则只需在http://server.runoob.com/server.php文件头部添加如下代码：

```
header('Access-Control-Allow-Origin:http://client.runoob.com');
```

允许多个域名跨域访问

指定多个域名（http://client1.runoob.com、http://client2.runoob.com等）跨域访问，则只需在http://server.runoob.com/server.php文件头部添加如下代码：

```
$origin = isset($_SERVER['HTTP_ORIGIN'])? $_SERVER['HTTP_ORIGIN'] :
 '';

$allow_origin = array(
    'http://client1.runoob.com',
    'http://client2.runoob.com'
);

if(in_array($origin, $allow_origin)){
    header('Access-Control-Allow-Origin:'.$origin);
}
```

允许所有域名跨域访问

允许所有域名访问则只需在http://server.runoob.com/server.php文件头部添加如下代码：

```
header('Access-Control-Allow-Origin:*');
```

CORS跨域支持post和get方式，而jsonp跨域只支持get方式

jwt 介绍

JSON Web Token（缩写 JWT）是目前最流行的跨域认证解决方案，本文介绍它的原理和用法。

一、跨域认证的问题

互联网服务离不开用户认证。一般流程是下面这样。

1、用户向服务器发送用户名和密码。

2、服务器验证通过后，在当前对话（session）里面保存相关数据，比如用户角色、登录时间等等。

3、服务器向用户返回一个 session_id，写入用户的 Cookie。

4、用户随后的每一次请求，都会通过 Cookie，将 session_id 传回服务器。

5、服务器收到 session_id，找到前期保存的数据，由此得知用户的身份。

这种模式的问题在于，扩展性（scaling）不好。单机当然没有问题，如果是服务器集群，或者是跨域的服务导向架构，就要求 session 数据共享，每台服务器都能够读取 session。

举例来说，A 网站和 B 网站是同一家公司的关联服务。现在要求，用户只要在其中一个网站登录，再访问另一个网站就会自动登录，请问怎么实现？

一种解决方案是 session 数据持久化，写入数据库或别的持久层。各种服务收到请求后，都向持久层请求数据。这种方案的优点是架构清晰，缺点是工程量比较大。另外，持久层万一挂了，就会单点失败。

另一种方案是服务器索性不保存 session 数据了，所有数据都保存在客户端，每次请求都发回服务器。JWT 就是这种方案的一个代表。

二、JWT 的原理

JWT 的原理是，服务器认证以后，生成一个 JSON 对象，发回给用户，就像下面这样。

```
1 {
```

```
  "姓名": "张三",
  "角色": "管理员",
  "到期时间": "2018年7月1日0点0分"
}
```

以后，用户与服务端通信的时候，都要发回这个 JSON 对象。服务器完全只靠这个对象认定用户身份。为了防止用户篡改数据，服务器在生成这个对象的时候，会加上签名（详见后文）。

服务器就不保存任何 session 数据了，也就是说，服务器变成无状态了，从而比较容易实现扩展。

三、JWT 的数据结构

实际的 JWT 大概就像下面这样。

它是一个很长的字符串，中间用点（.）分隔成三个部分。注意，JWT 内部是没有换行的，这里只是为了便于展示，将它写成了几行。

JWT 的三个部分依次如下。

```
Header（头部）
Payload（负载）
Signature（签名）
```

写成一行，就是下面的样子。

Header.Payload.Signature

下面依次介绍这三个部分。

3.1 Header

Header 部分是一个 JSON 对象，描述 JWT 的元数据，通常是下面的样子。

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

上面代码中，alg属性表示签名的算法（algorithm），默认是 HMAC SHA256（写成 HS256）；typ属性表示这个令牌（token）的类型（type），JWT 令牌统一写为JWT。

最后，将上面的 JSON 对象使用 Base64URL 算法（详见后文）转成字符串。

3.2 Payload

Payload 部分也是一个 JSON 对象，用来存放实际需要传递的数据。JWT 规定了7个官方字段，供选用。

```
iss (issuer)：签发人
exp (expiration time)：过期时间
sub (subject)：主题
aud (audience)：受众
nbf (Not Before)：生效时间
iat (Issued At)：签发时间
jti (JWT ID)：编号
```

除了官方字段，你还可以在这个部分定义私有字段，下面就是一个例子。

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

注意，JWT 默认是不加密的，任何人都可以读到，所以不要把秘密信息放在这个部分。

这个 JSON 对象也要使用 Base64URL 算法转成字符串。

3.3 Signature

Signature 部分是对前两部分的签名，防止数据篡改。

首先，需要指定一个密钥（secret）。这个密钥只有服务器才知道，不能泄露给用户。然后，使用 Header 里面指定的签名算法（默认是 HMAC SHA256），按照下面的公式产生签名。

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

算出签名以后，把 Header、Payload、Signature 三个部分拼成一个字符串，每个部分之间用"点"（.）分隔，就可以返回给用户。

3.4 Base64URL

前面提到，Header 和 Payload 串型化的算法是 Base64URL。这个算法跟 Base64 算法基本类似，但有一些小的不同。

JWT 作为一个令牌（token），有些场合可能会放到 URL（比如 api.example.com/?token=xxx）。Base64 有三个字符+、/和=，在 URL 里面有特殊含义，所以要被替换掉：=被省略、+替换成-，/替换成_ 。这就是 Base64URL 算法。

四、JWT 的使用方式

客户端收到服务器返回的 JWT，可以储存在 Cookie 里面，也可以储存在 localStorage。

此后，客户端每次与服务器通信，都要带上这个 JWT。你可以把它放在 Cookie 里面自动发送，但是这样不能跨域，所以更好的做法是放在 HTTP 请求的头信息Authorization字段里面。

Authorization: Bearer
另一种做法是，跨域的时候，JWT 就放在 POST 请求的数据体里面。

五、JWT 的几个特点

（1）JWT 默认是不加密，但也是可以加密的。生成原始 Token 以后，可以用密钥再加密一次。

（2）JWT 不加密的情况下，不能将秘密数据写入 JWT。

（3）JWT 不仅可以用于认证，也可以用于交换信息。有效使用 JWT，可以降低服务器查询数据库的次数。

（4）JWT 的最大缺点是，由于服务器不保存 session 状态，因此无法在使用过程中废止某个 token，或者更改 token 的权限。也就是说，一旦 JWT 签发了，在到期之前就会始终有效，除非服务器部署额外的逻辑。

（5）JWT 本身包含了认证信息，一旦泄露，任何人都可以获得该令牌的所有权限。为了减少盗用，JWT 的有效期应该设置得比较短。对于一些比较重要的权限，使用时应该再次对用户进行认证。

（6）为了减少盗用，JWT 不应该使用 HTTP 协议明码传输，要使用 HTTPS 协议传输。

参考地址：
jwt官网
jwt php开源项目地址

转自阮一峰的网络日志

未经授权获得用户个人信息

1、泄露用户个人信息，比如电话、住址等。

如某接口未做验证，别人可以通过随便传用户id获取用户的个人信息。获取用户信息的接口一定要做校验，只能授权用户可以获取。

2、将loginId和nickname分离

设计站点时要将loginId和nickname分开，登录id是用户的隐私信息，只有用户本人可以看到；而nickname不能用于登录，但可以公开给所有人看。防止攻击者拿用于登录的loginId进行暴力破解。

服务器漏洞

服务器漏洞

服务器除了以上提到的那些大名鼎鼎的漏洞和臭名昭著的攻击以外，其实还有很多其他的漏洞，往往也很容易被忽视，在这个小节也稍微介绍几种。

越权操作漏洞

如果你的系统是有登录控制的，那就要格外小心了，因为很有可能你的系统越权操作漏洞，越权操作漏洞可以简单的总结为 「A 用户能看到或者操作 B 用户的隐私内容」，如果你的系统中还有权限控制就更加需要小心了。所以每一个请求都需要做 userid 的判断

以下是一段有漏洞的后端示意代码：

```
// ctx 为请求的 context 上下文
let msgId = ctx.params.msgId;

mysql.query(
    'SELECT * FROM msg_table WHERE msg_id = ?',
    [msgId]
);
```

以上代码是任何人都可以查询到任何用户的消息，只要有 msg_id 就可以，这就是比较典型的越权漏洞，需要如下这么改进一下：

```
// ctx 为请求的 context 上下文
let msgId = ctx.params.msgId;
let userId = ctx.session.userId; // 从会话中取出当前登陆的 userId

mysql.query(
    'SELECT * FROM msg_table WHERE msg_id = ? AND user_id = ?',
    [msgId, userId]
);
```

嗯，大概就是这个意思，如果有更严格的权限控制，那在每个请求中凡是涉及到数据库的操作都需要先进行严格的验证，并且在设计数据库表的时候需要考虑进 userId 的账号关联以及权限关联。

目录遍历漏洞

目录遍历漏洞指通过在 URL 或参数中构造 ../，./ 和类似的跨父目录字符串的 ASCII 编码、unicode 编码等，完成目录跳转，读取操作系统各个目录下的敏感文件，也可以称作「任意文件读取漏洞」。

目录遍历漏洞原理：程序没有充分过滤用户输入的 ../ 之类的目录跳转符，导致用户可以通过提交目录跳转来遍历服务器上的任意文件。使用多个.. 符号，不断向上跳转，最终停留在根 /，通过绝对路径去读取任意文件。

目录遍历漏洞几个示例和测试，一般构造 URL 然后使用浏览器直接访问，或者使用 Web 漏洞扫描工具检测，当然也可以自写程序测试。

```
http://somehost.com/../../../../../../../../../etc/passwd
http://somehost.com/some/path?file=../../Windows/system.ini

# 借助 %00 空字符截断是一个比较经典的攻击手法
http://somehost.com/some/path?file=../../Windows/system.ini%00.js

# 使用了 IIS 的脚本目录来移动目录并执行指令
http://somehost.com/scripts/..%5c../Windows/System32/cmd.exe?/c+dir+c:\
```

防御 方法就是需要对 URL 或者参数进行 ../，./ 等字符的转义过滤。

物理路径泄漏

物理路径泄露属于低风险等级缺陷，它的危害一般被描述为「攻击者可以利用此漏洞得到信息，来对系统进一步地攻击」，通常都是系统报错 500 的错误信息直接返回到页面可见导致的漏洞。得到物理路径有些时候它能给攻击者带来一些有用的信息，比如说：可以大致了解系统的文件目录结构；可以看出系统所使用的第三方软件；也说不定会得到一个合法的用户名（因为很多人把自己的用户名作为网站的目录名）。

防止这种泄漏的方法就是做好后端程序的出错处理，定制特殊的 500 报错页面。

源码暴露漏洞

和物理路径泄露类似，就是攻击者可以通过请求直接获取到你站点的后端源代码，然后就可以对系统进一步研究攻击。那么导致源代码暴露的原因是什么呢？基本上就是发生在服务器配置上了，服务器可以设置哪些路径的文件才可以被直接访问的，这里给一个 koa 服务起的例子，正常的 koa 服务器可以通过 koa-static 中间件去指定静态资源的目录，好让静态资源可以通过路径的路由访问。比如你的系统源代码目录是这样的：

```
|- project
    |- src
    |- static
    |- ...
|- server.js
```

你想要将 static 的文件夹配成静态资源目录，你应该会在 server.js 做如下配置：

```
const Koa = require('koa');
const serve = require('koa-static');
const app = new Koa();

app.use(serve(__dirname + '/project/static'));
```

但是如果配错了静态资源的目录，可能就出大事了，比如：

```
// ...
app.use(serve(__dirname + '/project'));
```

这样所有的源代码都可以通过路由访问到了，所有的服务器都提供了静态资源机制，所以在通过服务器配置静态资源目录和路径的时候，一定要注意检验，不然很可能产生漏洞。

最后，希望 Web 开发者们能够管理好自己的代码隐私，注意代码安全问题，比如不要将产品的含有敏感信息的代码放到第三方外部站点或者暴露给外部用户，尤其是前端代码，私钥类似的保密性的东西不要直接输出在代码里或者页面中。也许还有很多值得注意的点，但是归根结底还是绷住安全那根弦，对待每一行代码都要多多推敲。

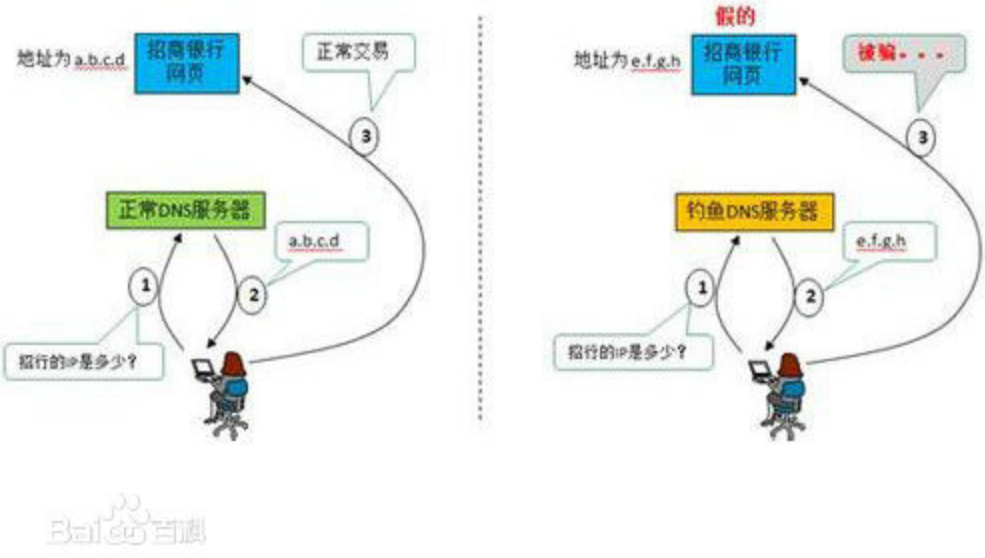
流量劫持

流量劫持

流量劫持应该算是黑产行业的一大经济支柱了吧？简直是让人恶心到吐，不吐槽了，还是继续谈干货吧，流量劫持基本分两种：DNS 劫持 和 HTTP 劫持，目的都是一样的，就是当用户访问 zoumiaojiang.com 的时候，给你展示的并不是或者不完全是 zoumiaojiang.com 提供的“内容”。比如访问的页面上被插入广告、弹窗等。

DNS 劫持

DNS 劫持，也叫做域名劫持，可以这么理解，「你打了一辆车想去商场吃饭，结果你打的车是小作坊派来的，直接给你拉到小作坊去了」，DNS 的作用是把网络地址域名对应到真实的计算机能够识别的 IP 地址，以便计算机能够进一步通信，传递网址和内容等。如果当用户通过某一个域名访问一个站点的时候，被篡改的 DNS 服务器返回的是一个恶意的钓鱼站点的 IP，用户就被劫持到了恶意钓鱼站点，然后继而会被钓鱼输入各种账号密码信息，泄漏隐私。



这类劫持，要不就是网络运营商搞的鬼，一般小的网络运营商与黑产勾结会劫持 DNS，要不就是电脑中毒，被恶意篡改了路由器的 DNS 配置，基本上做为开发者或站长却是很难察觉的，除非有用户反馈，现在升级版的 DNS 劫持还可以对特定用户、特定区域等使用了用户画像进行筛选用户劫持的办法，另外这

类广告显示更加随机更小，一般站长除非用户投诉否则很难觉察到，就算觉察到了取证举报更难。无论如何，如果接到有 DNS 劫持的反馈，一定要做好以下几件事：

> 取证很重要，时间、地点、IP、拨号账户、截屏、URL 地址等一定要有。
> 可以跟劫持区域的电信运营商进行投诉反馈。
> 如果投诉反馈无效，直接去工信部投诉，一般来说会加白你的域名。

HTTP 劫持

HTTP 劫持您可以这么理解，「你打了一辆车想去商场吃饭，结果司机跟你一路给你递小作坊的广告」，HTTP 劫持主要是当用户访问某个站点的时候会经过运营商网络，而不法运营商和黑产勾结能够截获 HTTP 请求返回内容，并且能够篡改内容，然后再返回给用户，从而实现劫持页面，轻则插入小广告，重则直接篡改成钓鱼网站页面骗用户隐私。能够实施流量劫持的根本原因，是 HTTP 协议没有办法对通信对方的身份进行校验以及对数据完整性进行校验。如果能解决这个问题，则流量劫持将无法轻易发生。所以防止 HTTP 劫持的方法只有将内容加密，让劫持者无法破解篡改，这样就可以防止 HTTP 劫持了。

HTTPS 协议就是一种基于 SSL 协议的安全加密网络应用层协议，可以很好的防止 HTTP 劫持。如果不想站点被 HTTP 劫持，赶紧将你的站点全站改造成 HTTPS 吧。

DDoS 攻击

DDoS 攻击

DDoS 又叫分布式拒绝服务，全称 Distributed Denial of Service，其原理就是利用大量的请求造成资源过载，导致服务不可用，这个攻击应该不能算是安全问题，这应该算是一个另类的存在，因为这种攻击根本就是耍流氓的存在，「伤敌一千，自损八百」的行为。出于保护 Web App 不受攻击的攻防角度，还是介绍一下 DDoS 攻击吧，毕竟也是挺常见的。

DDoS 攻击可以理解为：「你开了一家店，隔壁家点看不惯，就雇了一大堆黑社会人员进你店里干坐着，也不消费，其他客人也进不来，导致你营业惨淡」。为啥说 DDoS 是个「伤敌一千，自损八百」的行为呢？毕竟隔壁店还是花了不少钱雇黑社会但是啥也没得到不是？DDoS 攻击的目的基本上就以下几个：

深仇大恨，就是要干死你
敲诈你，不给钱就干你
忽悠你，不买我防火墙服务就会有“人”继续干你
也许你的站点遭受过 DDoS 攻击，具体什么原因怎么解读见仁见智。DDos 攻击从层次上可分为网络层攻击与应用层攻击，从攻击手法上可分为快型流量攻击与慢型流量攻击，但其原理都是**造成资源过载，导致服务不可用**。

网络层 DDoS

网络层 DDos 攻击包括 SYN Flood、ACK Flood、UDP Flood、ICMP Flood 等。

SYN Flood 攻击

SYN flood 攻击主要利用了 TCP 三次握手过程中的 Bug，我们都知道 TCP 三次握手过程是要建立连接的双方发送 SYN，SYN + ACK，ACK 数据包，而当攻击方随意构造源 IP 去发送 SYN 包时，服务器返回的 SYN + ACK 就不能得到应答（因为 IP 是随意构造的），此时服务器就会尝试重新发送，并且会有至少 30s 的等待时间，导致资源饱和服务不可用，此攻击属于慢型 DDoS 攻击。

ACK Flood 攻击

ACK Flood 攻击是在 TCP 连接建立之后，所有的数据传输 TCP 报文都是带有 ACK 标志位的，主机在接收到一个带有 ACK 标志位的数据包的时候，需要检查该数据包所表示的连接四元组是否存在，如果存

在则检查该数据包所表示的状态是否合法，然后再向应用层传递该数据包。如果在检查中发现该数据包不合法，例如该数据包所指向的目的端口在本机并未开放，则主机操作系统协议栈会回应 RST 包告诉对方此端口不存在。

UDP Flood 攻击

UDP flood 攻击是由于 UDP 是一种无连接的协议，因此攻击者可以伪造大量的源 IP 地址去发送 UDP 包，此种攻击属于大流量攻击。正常应用情况下，UDP 包双向流量会基本相等，因此发起这种攻击的攻击者在消耗对方资源的时候也在消耗自己的资源。

ICMP Flood 攻击

ICMP Flood 攻击属于大流量攻击，其原理就是不断发送不正常的 ICMP 包（所谓不正常就是 ICMP 包内容很大），导致目标带宽被占用，但其本身资源也会被消耗。目前很多服务器都是禁 ping 的（在防火墙在可以屏蔽 ICMP 包），因此这种攻击方式已经落伍。

网络层 DDoS 防御

网络层的 DDoS 攻击究其本质其实是无法防御的，我们能做得就是不断优化服务本身部署的网络架构，以及提升网络带宽。当然，还是做好以下几件事也是有助于缓解网络层 DDoS 攻击的冲击：

> 网络架构上做好优化，采用负载均衡分流。
> 确保服务器的系统文件是最新的版本，并及时更新系统补丁。
> 添加抗 DDos 设备，进行流量清洗。
> 限制同时打开的 SYN 半连接数目，缩短 SYN 半连接的 Timeout 时间。
> 限制单 IP 请求频率。
> 防火墙等防护设置禁止 ICMP 包等。
> 严格限制对外开放的服务器的向外访问。
> 运行端口映射程序或端口扫描程序，要认真检查特权端口和非特权端口。
> 关闭不必要的服务。
> 认真检查网络设备和主机/服务器系统的日志。只要日志出现漏洞或是时间变更,那这台机器就可能遭到了攻击。
> 限制在防火墙外与网络文件共享。这样会给黑客截取系统文件的机会，主机的信息暴露给黑客，无疑是给了对方入侵的机会。
> 加钱堆机器。。
> 报警。。

应用层 DDoS

应用层 DDoS 攻击不是发生在网络层，是发生在 TCP 建立握手成功之后，应用程序处理请求的时候，现在很多常见的 DDoS 攻击都是应用层攻击。应用层攻击千变万化，目的就是在网络应用层耗尽你的带宽，下面列出集中典型的攻击类型。

CC 攻击

当时绿盟为了防御 DDoS 攻击研发了一款叫做 Collapasar 的产品，能够有效的防御 SYN Flood 攻击。黑客为了挑衅，研发了一款 Challenge Collapasar 攻击工具（简称 CC）。

CC 攻击的原理，就是针对消耗资源比较大的页面不断发起不正常的请求，导致资源耗尽。因此在发送 CC 攻击前，我们需要寻找加载比较慢，消耗资源比较多的网页，比如需要查询数据库的页面、读写硬盘文件的等。通过 CC 攻击，使用爬虫对某些加载需要消耗大量资源的页面发起 HTTP 请求。

DNS Flood

DNS Flood 攻击采用的方法是向被攻击的服务器发送大量的域名解析请求，通常请求解析的域名是随机生成或者是网络世界上根本不存在的域名，被攻击的DNS 服务器在接收到域名解析请求的时候首先会在服务器上查找是否有对应的缓存，如果查找不到并且该域名无法直接由服务器解析的时候，DNS 服务器会向其上层 DNS 服务器递归查询域名信息。域名解析的过程给服务器带来了很大的负载，每秒钟域名解析请求超过一定的数量就会造成 DNS 服务器解析域名超时。

根据微软的统计数据，一台 DNS 服务器所能承受的动态域名查询的上限是每秒钟 9000 个请求。而我们知道，在一台 P3 的 PC 机上可以轻易地构造出每秒钟几万个域名解析请求，足以使一台硬件配置极高的 DNS 服务器瘫痪，由此可见 DNS 服务器的脆弱性。

HTTP 慢速连接攻击

针对 HTTP 协议，先建立起 HTTP 连接，设置一个较大的 Conetnt-Length，每次只发送很少的字节，让服务器一直以为 HTTP 头部没有传输完成，这样连接一多就很快会出现连接耗尽。

应用层 DDoS 防御

> 判断 User-Agent 字段（不可靠，因为可以随意构造）
> 针对 IP + cookie，限制访问频率（由于 cookie 可以更改，IP 可以使用代理，或者肉鸡，也不可靠）

> 关闭服务器最大连接数等，合理配置中间件，缓解 DDoS 攻击。
> 请求中添加验证码，比如请求中有数据库操作的时候。
> 编写代码时，尽量实现优化，并合理使用缓存技术，减少数据库的读取操作。
> 加钱堆机器。。
> 报警。。
> 应用层的防御有时比网络层的更难，因为导致应用层被 DDoS 攻击的因素非常多，有时往往是因为程序员的失误，导致某个页面加载需要消耗大量资源，有时是因为中间件配置不当等等。而应用层 DDoS 防御的核心就是区分人与机器（爬虫），因为大量的请求不可能是人为的，肯定是机器构造的。因此如果能有效的区分人与爬虫行为，则可以很好地防御此攻击。

####其他 DDoS 攻击
发起 DDoS 也是需要大量的带宽资源的，但是互联网就像森林，林子大了什么鸟都有，DDoS 攻击者也能找到其他的方式发起廉价并且极具杀伤力的 DDoS 攻击。

利用 XSS

举个例子，如果 12306 页面有一个 XSS 持久型漏洞被恶意攻击者发现，只需在春节抢票期间在这个漏洞中执行脚本使得往某一个小站点随便发点什么请求，然后随着用户访问的增多，感染用户增多，被攻击的站点自然就会迅速瘫痪了。这种 DDoS 简直就是无本万利，不用惊讶，现在大站有 XSS 漏洞的不要太多。

来自 P2P 网络攻击

大家都知道，互联网上的 P2P 用户和流量都是一个极为庞大的数字。如果他们都去一个指定的地方下载数据，成千上万的真实 IP 地址连接过来，没有哪个设备能够支撑住。拿 BT 下载来说，伪造一些热门视频的种子，发布到搜索引擎，就足以骗到许多用户和流量了，但是这只是基础攻击。
高级的 P2P 攻击，是直接欺骗资源管理服务器。如迅雷客户端会把自己发现的资源上传到资源管理服务器，然后推送给其它需要下载相同资源的用户，这样，一个链接就发布出去。通过协议逆向，攻击者伪造出大批量的热门资源信息通过资源管理中心分发出去，瞬间就可以传遍整个 P2P 网络。更为恐怖的是，这种攻击是无法停止的，即使是攻击者自身也无法停止，攻击一直持续到 P2P 官方发现问题更新服务器且下载用户重启下载软件为止。

最后总结下，DDoS 不可能防的住，就好比你的店只能容纳 50 人，黑社会有 100 人，你就换一家大店，能容纳 500 人，然后黑社会又找来了 1000 人，这种堆人头的做法就是 DDoS 本质上的攻防之道，「道高一尺，魔高一丈，魔高一尺，道高一丈」，讲真，必要的时候就答应勒索你的人的条件吧，实在不行就报警吧。

命令行注入

命令行注入

命令行注入漏洞，指的是攻击者能够通过 HTTP 请求直接侵入主机，执行攻击者预设的 shell 命令，听起来好像匪夷所思，这往往是 Web 开发者最容易忽视但是却是最危险的一个漏洞之一，看一个实例：

假如现在需要实现一个需求：用户提交一些内容到服务器，然后在服务器执行一些系统命令去产出一个结果返回给用户，接口的部分实现如下：

```
// 以 Node.js 为例，假如在接口中需要从 github 下载用户指定的 repo
const exec = require('mz/child_process').exec;
let params = {/* 用户输入的参数 */};

exec(`git clone ${params.repo} /some/path`);
```

这段代码确实能够满足业务需求，正常的用户也确实能从指定的 git repo 上下载到想要的代码，可是和 SQL 注入一样，这段代码在恶意攻击者眼中，简直就是香饽饽。

如果 params.repo 传入的是 `https://github.com/zoumiaojiang/zoumiaojiang.github.io.git` 当然没问题了。
可是如果 params.repo 传入的是`https://github.com/xx/xx.git && rm -rf /* &&`恰好你的服务是用 root 权限起的就惨了。

具体恶意攻击者能用命令行注入干什么也像 SQL 注入一样，手法是千变万化的，比如「反弹 shell 注入」等，但原理都是一样的，我们绝对有能力防止命令行注入发生。

命令行注入预防：

```
后端对前端提交内容需要完全选择不相信，并且对其进行规则限制（比如正则表达式）。
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。
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。
```

还是前面的例子，我们可以做到如下：

```
const exec = require('mz/child_process').exec;

// 借助 shell-escape npm 包解决参数转义过滤问题
const shellescape = require('shell-escape');

let params = {/* 用户输入的参数 */};

// 先过滤一下参数，让参数符合预期
if (!/正确的表达式/.test(params.repo)) {
    return;
}

let cmd = shellescape([
    'git',
    'clone',
    params.repo,
    '/some/path'
]);

// cmd 的值: git clone 'https://github.com/xx/xx.git && rm -rf / &&' /some/path
// 这样就不会被注入成功了。
exec(cmd);
```

无论是在何种后端语言环境中，凡是涉及到代码调用系统 shell 命令的时候都一定要谨慎。

SQL 注入

SQL 注入

SQL 注入漏洞（SQL Injection）是 Web 开发中最常见的一种安全漏洞。可以用它来从数据库获取敏感信息，或者利用数据库的特性执行添加用户，导出文件等一系列恶意操作，甚至有可能获取数据库乃至系统用户最高权限。

而造成 SQL 注入的原因是因为程序没有有效的转义过滤用户的输入，使攻击者成功的向服务器提交恶意的 SQL 查询代码，程序在接收后错误的将攻击者的输入作为查询语句的一部分执行，导致原始的查询逻辑被改变，额外的执行了攻击者精心构造的恶意代码。

很多 Web 开发者没有意识到 SQL 查询是可以被篡改的，从而把 SQL 查询当作可信任的命令。殊不知，SQL 查询是可以绕开访问控制，从而绕过身份验证和权限检查的。更有甚者，有可能通过 SQL 查询去运行主机系统级的命令。

SQL 注入原理

下面将通过一些真实的例子来详细讲解 SQL 注入的方式的原理。

考虑以下简单的管理员登录表单：

```
<form action="/login" method="POST">
    <p>Username: <input type="text" name="username" /></p>
    <p>Password: <input type="password" name="password" /></p>
    <p><input type="submit" value="登陆" /></p>
</form>
```

后端的 SQL 语句可能是如下这样的：

```
let querySQL = `
    SELECT *
    FROM user
    WHERE username='${username}'
    AND psw='${password}'
```

```
6 `;
```

接下来就是执行 sql 语句...
目的就是来验证用户名和密码是不是正确，按理说乍一看上面的 SQL 语句也没什么毛病，确实是能够达到我们的目的，可是你只是站在用户会老老实实按照你的设计来输入的角度来看问题，如果有一个恶意攻击者输入的用户名是 zoumiaojiang' OR 1 = 1 --，密码随意输入，就可以直接登入系统了。WFT!

冷静下来思考一下，我们之前预想的真实 SQL 语句是:

```
1 SELECT * FROM user WHERE username='zoumiaojiang' AND psw='mypassword'
```

可以恶意攻击者的奇怪用户名将你的 SQL 语句变成了如下形式:

```
1 SELECT * FROM user WHERE username='zoumiaojiang' OR 1 = 1 --' AND psw
  ='xxxx'
```

在 SQL 中，-- 是注释后面的内容的意思，所以查询语句就变成了:

```
1 SELECT * FROM user WHERE username='zoumiaojiang' OR 1 = 1
```

这条 SQL 语句的查询条件永远为真，所以意思就是恶意攻击者不用我的密码，就可以登录进我的账号，然后可以在里面为所欲为，然而这还只是最简单的注入，牛逼的 SQL 注入高手甚至可以通过 SQL 查询去运行主机系统级的命令，将你主机里的内容一览无余，这里我也没有这个能力讲解的太深入，毕竟不是专业研究这类攻击的，但是通过以上的例子，已经了解了 SQL 注入的原理，我们基本已经能找到防御 SQL 注入的方案了。

如何预防 SQL 注入

防止 SQL 注入主要是不能允许用户输入的内容影响正常的 SQL 语句的逻辑，当用户的输入的信息将要用来拼接 SQL 语句的话，我们应该永远选择不相信，任何内容都必须进行转义过滤，当然做到这个还是不够的，下面列出防御 SQL 注入的几点注意事项:

> 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

> 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。

> 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。

> 所有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。例如 Node.js 中的 mysqljs 库的 query 方法中的 ？占位参数。(预处理机制)

> 在应用发布之前建议使用专业的 SQL 注入检测工具进行检测，以及时修补被发现的 SQL 注入漏洞。网上有很多这方面的开源工具，例如 sqlmap、SQLninja 等。

> 避免网站打印出 SQL 错误信息，比如类型错误、字段不匹配等，把代码里的 SQL 语句暴露出来，以防止攻击者利用这些错误信息进行 SQL 注入。

> 不要过于细化返回的错误信息，如果目的是方便调试，就去使用后端日志，不要在接口上过多的暴露出错信息，毕竟真正的用户不关心太多的技术细节，只要话术合理就行。

> 碰到要操作的数据库的代码，一定要慎重，小心使得万年船，多找几个人多来几次 code review，将问题都暴露出来，而且要善于利用工具，操作数据库相关的代码属于机密，没事不要去各种论坛晒自家站点的 SQL 语句，万一被人盯上了呢?

XSS跨站脚本攻击

XSS

XSS (Cross Site Script)，跨站脚本攻击，因为缩写和 CSS (Cascading Style Sheets) 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。XSS 的攻击方式千变万化，但还是可以大致细分为几种类型。

非持久型 XSS

非持久型 XSS 漏洞，也叫反射型 XSS 漏洞，一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。

举一个例子，比如你的 Web 页面中包含有以下代码：

```
<select>
    <script>
        document.write('' 
            + '<option value=1>'
```

```
            +     location.href.substring(location.href.indexOf('default=') + 8)
            + '</option>'
        );
        document.write('<option value=2>English</option>');
    </script>
</select>
```

攻击者可以直接通过 URL（类似：`https://xx.com/xx?default=<script>alert(document.cookie)</script>`）注入可执行的脚本代码。

非持久型 XSS 漏洞攻击特征：

```
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。

攻击者需要诱骗点击
反馈率低，所以较难发现和响应修复
盗取用户敏感保密信息
```

非持久型 XSS 漏洞预防：

> Web 页面渲染的所有内容或者渲染的数据都必须来自于服务端。

> 尽量不要从 URL，document.referrer，document.forms 等这种 DOM API 中获取数据直接渲染。
> 尽量不要使用 eval, new Function()，document.write()，document.writeln(),window.setInterval()，window.setTimeout()，innerHTML，document.creteElement() 等可执行字符串的方法。

> 如果做不到以上几点，也必须对涉及 DOM 渲染的方法传入的字符串参数做 escape 转义。
> 前端渲染的时候对任何的字段都需要做 escape 转义编码。
> escape 转义的目的是将一些构成 HTML 标签的元素转义，比如 <，>，空格 等，转义成 <，>， 等显示转义字符。有很多开源的工具可以协助我们做 escape 转义。

持久型 XSS

持久型 XSS 漏洞，也被称为存储型 XSS 漏洞，一般存在于 Form 表单提交等交互功能，如发帖留言，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。

主要注入页面方式和非持久型 XSS 漏洞类似，只不过持久型的不是来源于 URL，refferer，forms 等，而是来源于后端从数据库中读出来的数据。持久型 XSS 攻击不需要诱骗点击，黑客只需要在提交表单的地方完成注入即可，但是这种 XSS 攻击的成本相对还是很高。

持久型 XSS 特点：

```
1 持久性，植入在数据库中
2 危害面广，甚至可以让用户机器变成 DDoS 攻击的肉鸡。
3 盗取用户敏感私密信息
```

持久型 XSS 漏洞预防：

```
1 后端在入库前应该选择不相信任何前端数据，将所有的字段统一进行转义处理。
2 后端在输出给前端数据统一进行转义处理。
3 前端在渲染页面 DOM 的时候应该选择不相信任何后端数据，任何字段都需要做转义处理。
```

基于字符集的 XSS

其实现在很多的浏览器以及各种开源的库都专门针对了 XSS 进行转义处理，尽量默认抵御绝大多数 XSS 攻击，但是还是有很多方式可以绕过转义规则，让人防不胜防。比如「基于字符集的 XSS 攻击」就是绕过这些转义处理的一种攻击方式，比如有些 Web 页面字符集不固定，用户输入非期望字符集的字符，有时会绕过转义过滤规则。

以基于 utf-7 的 XSS 为例
utf-7 是可以将所有的 unicode 通过 7bit 来表示的一种字符集 (但现在已经从 Unicode 规格中移除)。这个字符集为了通过 7bit 来表示所有的文字, 除去数字和一部分的符号,其它的部分将都以 base64 编码为基础的方式呈现。

```
1 <script>alert("xss")</script>
```

```
2 可以被解释为：
3 +ADw-script+AD4-alert(+ACI-xss+ACI-)+ADw-/script+AD4-
```

可以形成「基于字符集的 XSS 攻击」的原因是由于浏览器在 meta 没有指定 charset 的时候有自动识别编码的机制，所以这类攻击通常就是发生在没有指定或者没来得及指定 meta 标签的 charset 的情况下。

基于字符集的 XSS预防：

```
1 记住指定 <meta charset="utf-8">
2 XML 中不仅要指定字符集为 utf-8，而且标签要闭合
```

基于 Flash 的跨站 XSS

基于 Flash 的跨站 XSS 也是属于反射型 XSS 的一种，虽然现在开发 ActionScript 的产品线几乎没有了，但还是提一句吧，AS 脚本可以接受用户输入并操作 cookie，攻击者可以配合其他 XSS（持久型或者非持久型）方法将恶意 swf 文件嵌入页面中。主要是因为 AS 有时候需要和 JS 传参交互，攻击者会通过恶意的 XSS 注入篡改参数，窃取并操作cookie。

避免方法：

```
1 严格管理 cookie 的读写权限
2 对 Flash 能接受用户输入的参数进行过滤 escape 转义处理
```

未经验证的跳转 XSS

有一些场景是后端需要对一个传进来的待跳转的 URL 参数进行一个 302 跳转，可能其中会带有一些用户的敏感（cookie）信息。如果服务器端做302 跳转，跳转的地址来自用户的输入，攻击者可以输入一个恶意的跳转地址来执行脚本。

预防措施：

```
1 对待跳转的 URL 参数做白名单或者某种规则过滤
```

```
2 后端注意对敏感信息的保护, 比如 cookie 使用来源验证。
```

CSRF跨站请求伪造攻击

CSRF

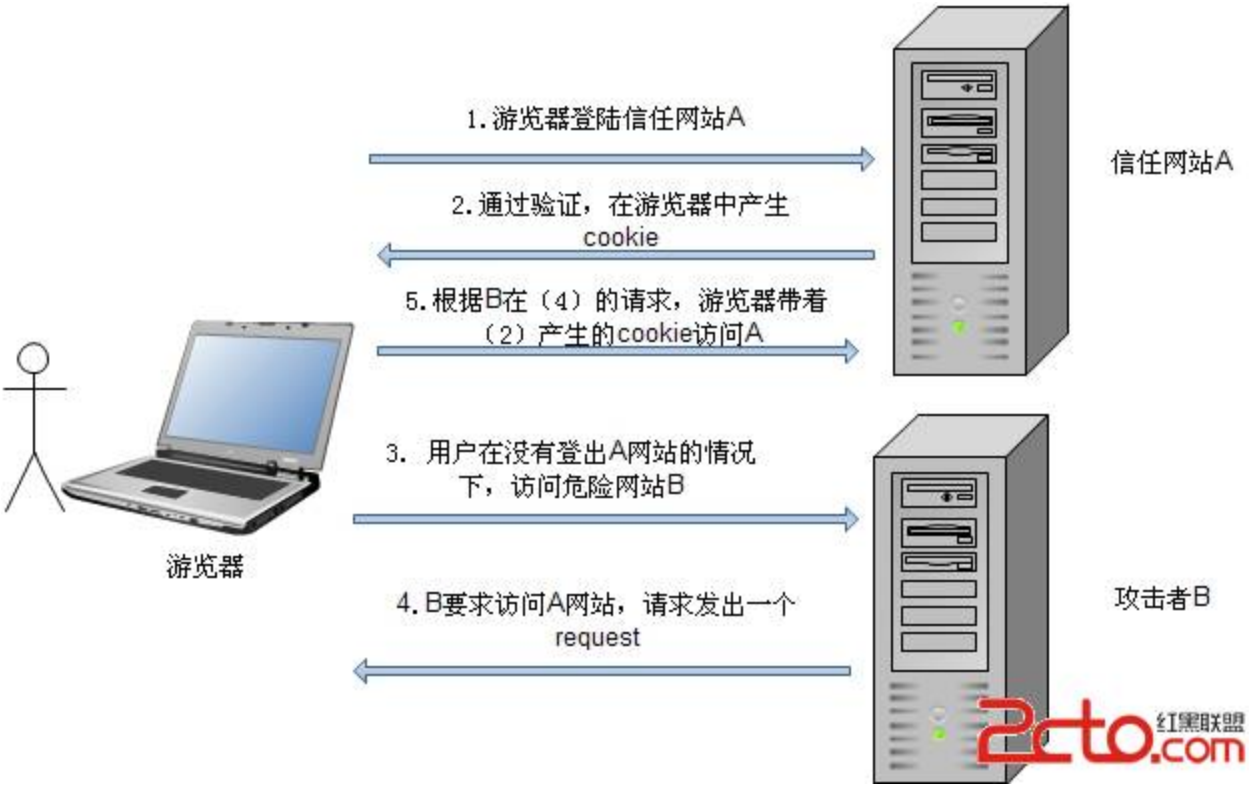
CSRF（Cross-Site Request Forgery），中文名称：跨站请求伪造攻击

理解： 攻击者可以盗用你的登陆信息，以你的身份模拟发送各种请求。攻击者只要借助少许的社会工程学的诡计，例如通过 QQ 等聊天软件发送的链接(有些还伪装成短域名，用户无法分辨)，攻击者就能迫使 Web 应用的用户去执行攻击者预设的操作。例如，当用户登录网络银行去查看其存款余额，在他没有退出时，就点击了一个 QQ 好友发来的链接，那么该用户银行帐户中的资金就有可能被转移到攻击者指定的帐户中。

所以遇到 CSRF 攻击时，将对终端用户的数据和操作指令构成严重的威胁。当受攻击的终端用户具有管理员帐户的时候，CSRF 攻击将危及整个 Web 应用程序。

CSRF 原理

下图大概描述了 CSRF 攻击的原理，可以理解为有一个小偷在你配钥匙的地方得到了你家的钥匙，然后拿着要是去你家想偷什么偷什么。



完成 CSRF 攻击必须要有三个条件：

```
用户已经登录了站点 A，并在本地记录了 cookie

在用户没有登出站点 A 的情况下（也就是 cookie 生效的情况下），
访问了恶意攻击者提供的引诱危险站点 B （B 站点要求访问站点A）。

站点 A 没有做任何 CSRF 防御
```

预防 CSRF

CSRF 的防御可以从服务端和客户端两方面着手，防御效果是从服务端着手效果比较好，现在一般的 CSRF 防御也都在服务端进行。服务端的预防 CSRF 攻击的方式方法有多种，但思路上都是差不多的，主要从以下两个方面入手：

```
正确使用 GET，POST 请求和 cookie

在非 GET 请求中增加 token

为每个用户生成一个唯一的 cookie token，所有表单都包含同一个伪随机值，这种方案最简单，
因为攻击者不能获得第三方的 cookie(理论上)，所以表单中的数据也就构造失败，
但是由于用户的 cookie 很容易由于网站的 XSS 漏洞而被盗取，所以这个方案必须要在没有 XSS 的情况下才安全。

每个 POST 请求使用验证码，这个方案算是比较完美的，但是需要用户多次输入验证码，
用户体验比较差，所以不适合在业务中大量运用。

渲染表单的时候，为每一个表单包含一个 csrfToken，提交表单的时候，
带上 csrfToken，然后在后端做 csrfToken 验证。
```

CSRF 的防御可以根据应用场景的不同自行选择。CSRF 的防御工作确实会在正常业务逻辑的基础上带来很多额外的开发量，但是这种工作量是值得的，毕竟用户隐私以及财产安全是产品最基础的根本。

Web 页面人机识别验证的探索与实践

在电商行业，线上的营销活动特别多。在移动互联网时代，一般为了活动的快速上线和内容的即时更新，大部分的业务场景仍然通过 Web 页面来承载。但由于 Web 页面天生“环境透明”，相较于移动客户端页面在安全性上存在更大的挑战。本文主要以移动端 Web 页面为基础来讲述如何提升页面安全性。

活动 Web 页面的安全挑战

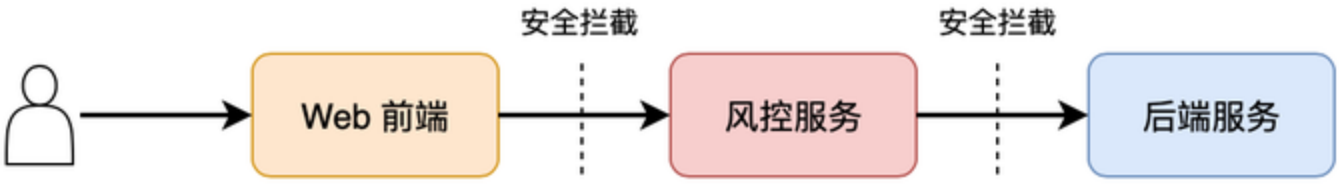
对于营销活动类的 Web 页面，领券、领红包、抽奖等活动方式很常见。此类活动对于普通用户来说大多数时候就是“拼手气”，而对于非正常用户来说，可以通过直接刷活动 API 接口的“作弊”方式来提升“手气”。这样的话，对普通用户来说，就变得很不公平。

对于活动运营的主办方来说，如果风控措施做的不好，这类刷接口的“拼手气”方式可能会对企业造成较大的损失。如本来计划按 7 天发放的红包，在上线 1 天就被刷光了，活动的营销成本就会被意外提升。主办方想发放给用户的满减券、红包，却大部分被黄牛使用自动脚本刷走，而真正想参与活动的用，却无法享受活动优惠。

终端用户到底是人还是机器，网络请求是否为真实用户发起，是否存在安全漏洞并且已被“羊毛党”恶意利用等等，这些都是运营主办方要担心的问题。

安全防范的基本流程

为了提升活动 Web 页面的安全性，通常会引入专业的风控服务。引入风控服务后，安全防护的流程大致如图所示。



Web 前端：

用户通过 Web 页面来参与活动，同时 **Web 前端也会收集用于人机识别验证的用户交互行为数据**。由于不同终端（移动端 H5 页面和 PC 端页面）交互形式不同，收集用户交互行为数据的侧重点也会有所不同。

风控服务：

一般大公司都会有专业的风控团队来提供风控服务，在美团内部有**智能反爬系统来基于地理位置、IP地址等大数据来提供频次限制、黑白名单限制**等常规的基础风控拦截服务。甚至还有**依托于海量的全业务场景的用户大数据，使用贝叶斯模型、神经网络等来构建专业度较深的服务**。风控服务可以为 Web 前端提供**通用的独立验证 SDK：验证码、滑块验证等区分人机的“图灵验证”，也可以为服务端提供 Web API 接口的验证等。**

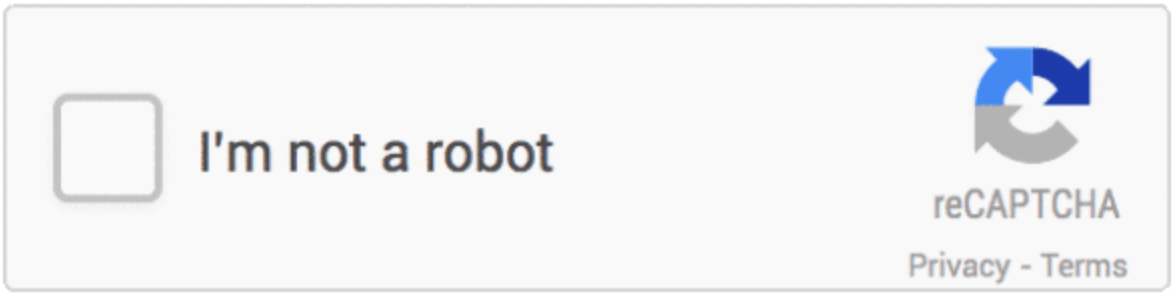
后端业务服务：

负责处理活动业务逻辑，如给用户发券、发红包，处理用户抽奖等。请求需要经过风控服务的验证，确保其安全性，然后再来处理实际业务逻辑，通常，在处理完实际业务逻辑时，还会有针对业务本身的风控防范。

对于活动 Web 页面来说，加入的风控服务主要为了做人机识别验证。在人机识别验证的专业领域上，我们可以先看看业界巨头 Google 是怎么做的。

Google 如何处理人机验证

Google 使用的人机验证服务是著名的 reCAPTCHA（Completely Automated Public Turing Test To Tell Computers and Humans Apart，区分人机的全自动图灵测试系统），也是应用最广的验证码系统。如今的 reCAPTCHA 已经不再需要人工输入难以识别的字符，它会检测用户的终端环境，追踪用户的鼠标轨迹，只需用户点击“我不是机器人”就能进行人机验证.



reCAPTCHA 的验证方式从早先的输入字符到现在的轻点按钮，在用户体验上，有了较大的提升。

而在活动场景中引入人机识别验证，如果只是简单粗暴地增加验证码，或者只是像 reCAPTCHA 那样增加点击“我不是机器人”的验证，都会牺牲用户体验，降低用户参加活动的积极性。

人机识别验证的技术挑战

理想的方案是在用户无感知的情况下做人机识别验证，这样既确保了安全又对用户体验无损伤。（若不可考虑这个因素，图片验证码 手机验证码等校验手段也可。）

从实际的业务场景出发再结合 Web 本身的环境，如果想实现理想的方案，可能会面临如下的技术挑战：

（1）需要根据用户的使用场景来定制人机识别验证的算法：Web 前端负责收集、上报用户交互行为数据，风控服务端校验上报的数据是否符合正常的用户行为逻辑。
（2）确保 Web 前端和风控服务端之间通信和数据传输的安全性。
（3）确保上述两大挑战中提到的逻辑和算法不会被代码反编译来破解。
在上述的三个挑战中，（1）已经实现了人机识别验证的功能，而（2）和（3）都是为了确保人机识别验证不被破解而做的安全防范。接下来，本文会分别针对这三个技术挑战来说明如何设计技术方案。

挑战一：根据用户使用场景来定制人机识别验证算法

先来分析一下用户的使用场景，正常用户参与活动的步骤是用户进入活动页面后，会有短暂的停留，然后点击按钮参与活动。这里所说的“参与活动”，最终都会在活动页面发起一个接口的请求。如果是非正常用户，可以直接跳过以上的实际动作而去直接请求参与活动的接口。

那么区别于正常用户和非正常用户就是那些被跳过的动作，对实际动作进一步归纳如下：

```
1 进入页面。
2 短暂的停留。
3 滚动页面。
4 点击按钮。
```

以上的动作又可以分为必需的操作和可选的操作。对这一连串动作产生的日志数据进行收集，在请求参与活动的接口时，将这些数据提交至后端，验证其合法性。这就是一个简单的人机识别验证。

在验证动作的合法性时，需要考虑到这些动作数据是不是能被轻易模拟。另外，动作的发生应该有一条时间线，可以给每个动作都增加一个时间戳，比如点击按钮肯定是在进入页面之后发生的。

一些特定的动作的日志数据也会有合理的区间，进入页面的动作如果以 JS 资源加载的时间为基准，那么加载时间可能大于 100 毫秒，小于 5 秒。而对于移动端的按钮点击，点击时记录的坐标值也会有对应的合理区间，这些合理的区间会根据实际的环境和情况来进行设置。

除此之外，**设备环境的数据也可以进行收集，包括用户参与活动时使用的终端类型、浏览器的类型、浏览器是否为客户端的容器等，如果使用了客户端，客户端是否会携带特殊的标识(UA校验)**等。

最后，还可以收集一些“无效”的数据，这些数据用于障人耳目，验证算法会将其忽略。尽管收集数据的动作是透明的，但是验证数据合法性不是透明的，攻击者无法知道，验证的算法中怎么区分哪些是有效、哪些是无效。这已经有点**“蜜罐数据”**的意思了。

挑战二：确保通信的安全性

收集的敏感数据要发送给风控服务端，进而确保通信过程的安全。

Web API 接口不能被中途拦截和篡改，**通信协议使用 HTTPS 是最基本的要求**；同时还要**让服务端生成唯一的 Token，在通信过程中都要携带该 Token**。
接口携带的敏感数据不能是明文的，敏感数据要进行加密，这样攻击者无法通过网络抓包来详细了解敏感数据的内容。

Token 的设计

Token 是一个简短的字符串，主要为了确保通信的安全。用户进入活动 Web 页面后，请求参与活动的接口之前，会从服务端获取 Token。该 **Token 的生成算法要确保 Token 的唯一性，**通过接口或 Cookie 传递给前端，**然后，前端在真正请求参与活动的接口时需要带上该 Token，风控服务端需要验证 Token 的合法性。**也就是说，Token 由服务端生成，传给前端，前端再原封不动的回传给服务端。一旦加入了 Token 的步骤，攻击者就不能直接去请求参与活动的接口了。

Token 由风控服务端基于用户的身份，根据一定的算法来生成，无法伪造，为了提升安全等级，Token 需要具有时效性，比如 10 分钟。可以使用 Redis 这类缓存服务来存储 Token，使用用户身份标识和 Token 建立 KV 映射表，并设置过期时间为 10 分钟（活动中token可以用完即删，保证每次参加活动token不同）。

虽然前端在 Cookie 中可以获取到 Token，但是前端不能对 Token 做持久化的缓存。一旦在 Cookie 中获取到了 Token，那么前端可以立即从 Cookie 中删除该 Token，这样能尽量确保 Token 的安全性和时效性。Token 存储在 Redis 中，也不会因为用户在参与活动时频繁的切换页面请求，而对服务造成太大的压力。

另外，Token 还可以有更多的用处：

```
1 标识参与活动用户的有效性。
2 敏感数据对称加密时生成动态密钥。
```

```
3 API 接口的数字签名。
4 敏感数据加密
```

通信时，传递的敏感数据可以使用常见的对称加密算法进行加密。

为了提升加密的安全等级，加密时的密钥可以动态生成，前端和风控服务端约定好动态密钥的生成规则即可。加密的算法和密钥也要确保不被暴露。

通过对敏感数据加密，攻击者在不了解敏感数据内容的前提下就更别提模拟构造请求内容了。

挑战三：化解纸老虎的尴尬

有经验的 Web 开发者看到这里，可能已经开始质疑了：在透明的前端环境中折腾安全不是白折腾吗？这就好比费了很大的劲却只是造了一个“纸老虎”，质疑是有道理的，但是且慢，通过一些安全机制的加强是可以让“纸老虎”尽可能的逼真。

本文一再提及的 Web 环境的透明性，是因为在实际的生产环境中的问题：前端的代码在压缩后，通过使用浏览器自带的格式化工具和断点工具，仍然具备一定的可读性，花点时间仍然可以理解代码的逻辑，这就给攻击者提供了大好的代码反编译机会。

如果要化解“纸老虎”的尴尬，就要对前端的代码进行混淆。

前端代码混淆

前端的 JS 代码压缩工具基本都是对变量、函数名称等进行缩短，压缩对于混淆的作用是比较弱。除了对代码进行压缩，还需要进行专门的混淆。

对代码进行混淆可以降低可读性，混淆工具有条件的话最好自研，开源的工具要慎用。或者基于 Uglify.js 来自定义混淆的规则，混淆程度越高可读性就越低。

代码混淆也需要把握一个度，太复杂的混淆可能会让代码无法运行，也有可能会影响本身的执行效率。同时还需要兼顾混淆后的代码体积，混淆前后的体积不能有太大的差距，合理的混淆程度很重要。

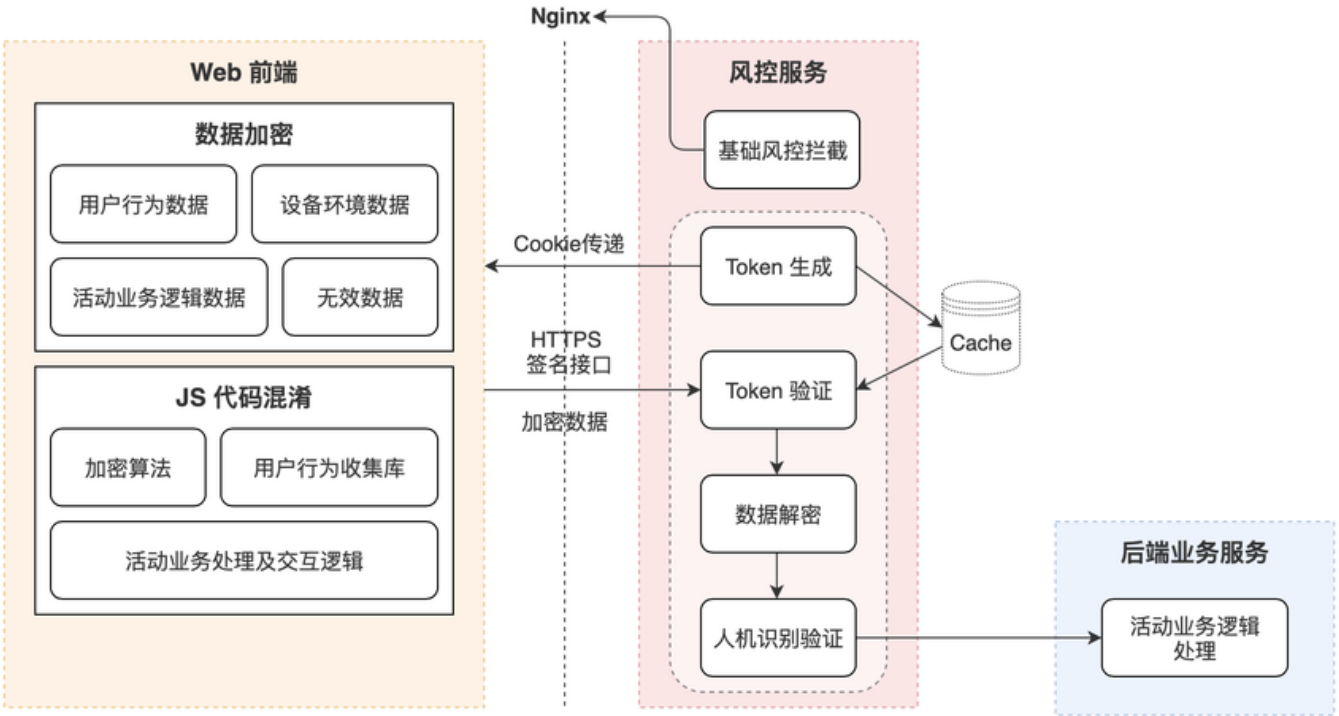
断点工具的防范会更麻烦些。在使用断点工具时通常都会导致代码延迟执行，而正常的业务逻辑都会立即执行，这是一个可以利用的点，可以考虑在代码执行间隔上来防范断点工具。

通过代码混淆和对代码进行特殊的处理，可以让格式化工具和断点工具变得没有用武之地。唯一有些小遗憾，就是处理后的代码也不能正常使用 Source Map 的功能了。

有了代码混淆，反编译的成本会非常高，这样“纸老虎”已经变得很逼真了。

技术方案设计

在讲解完如何解决关键的技术挑战后，就可以把相应的方案串起来，然后设计成一套可以实施的技术方案了。相对理想的技术方案架构图如下：



下面会按步骤来讲解技术方案的处理流程：

Step 0 基础风控拦截

基础风控拦截是上面提到的频次、名单等的拦截限制，在 Nginx 层就能直接实施拦截。如果发现是恶意请求，直接将请求过滤返回 403，这是初步的拦截，用户在请求 Web 页面的时候就开始起作用了。

Step 1 风控服务端生成 Token 后传给前端

Step 0 可能还没进入到活动 Web 页面，进入活动 Web 页面后才真正开始人机识别验证的流程，前端会先开始获取 Token。

Step 2 前端生成敏感数据

敏感数据应包含用户交互行为数据、设备环境数据、活动业务逻辑数据以及无效数据。

Step 3 使用 HTTPS 的签名接口发送数据

Token 可以作为 Authorization 的值添加到 Header 中，数据接口的签名可以有效防止 CSRF 的攻击。

Step 4 数据接口的校验

风控服务端收到请求后，会先验证数据接口签名中的 Token 是否有效。验证完 Token，才会对敏感数据进行解密。数据解密成功，再进一步对人机识别的数据合法性进行校验。

Step 5 业务逻辑的处理

前面的步骤为了做人机识别验证，这些验证不涉及到业务逻辑。在所有这些验证都通过后，后端业务服务才会开始处理实际的活动业务逻辑。处理完活动业务逻辑，最终才会返回用户参与活动的结果。

总结

为了提升活动 Web 页面的安全性，使用了各种各样的技术方案，我们将这些技术方案组合起来才能发挥安全防范的作用，如果其中某个环节处理不当，都可能会被当作漏洞来利用，从而导致整个验证方案被攻破。

为了验证技术方案的有效性，可以持续观察活动 API 接口的请求成功率。从请求成功率的数据中进一步分析“误伤”和“拦截”的数据，以进一步确定是否要对方案进行调优。

通过上述的人机识别验证的组合方案，可以大幅提升活动 Web 页面的安全性。在活动 Web 页面应作为一个标准化的安全防范流程，除了美团，像淘宝和天猫也有类似的流程。由于活动运营的环节和方法多且复杂，仅仅提升了 Web 页面也不敢保证就是铁板一块，安全需要关注的环节还很多，安全攻防是一项长期的“拉锯升级战”，安全防范措施也需要持续地优化升级。

参考资料

https://tech.meituan.com/2019/03/07/humans-and-bots-apart-for-activity-web-security.html

威胁&风险分析的STRIDE模型和DREAD模型

对于安全评估过程中的威胁分析有一些比较科学的方法，下面介绍微软的STRIDE模型和DREAD模型，用来分析哪些方面可能有威胁存在。

STRIDE模型

最早由微软提出来的，包含6个单词，我们在进行威胁分析时可以从以下6个方面去考虑：

威胁	定义	对应的安全属性
Spoofing（伪装）	冒充他人身份	认证
Tampering（篡改）	修改数据或代码	完整性
Repudiation（抵赖）	否认做过的事	不可抵赖性
Information Disclosure（信息泄露）	机密信息泄露	机密性
Denial of Service（拒绝服务）	拒绝服务	可用性
Elevation of Privilege（提升权限）	未经授权获得许可	授权

DREAD模型

由微软提出的，DREAD也是几个单词的首字母缩写，它指导我们应该从哪些方面判断一个威胁的风险程度。

等级	高（3）	中（2）	低（1）
Damage Potential	获取完全验证权限，执行管理员操作，非法上传文件	泄露敏感信息	泄露其他信息
Reproducibility	攻击者可以随意再次攻击	攻击者可以重复攻击，但有时间限制	攻击者很难重复攻击过程
Exploitability	初学者短期能掌握攻击方法	熟练的攻击者才能完成这次攻击	漏洞利用条件非常苛刻
Affected users	所有用户，默认配置，关键用户	部分用户，非默认配置	极少数用户，匿名用户
Discoverability	漏洞很显眼，攻击条件很容易获得	在私有区域，部分人能看到，需要深入挖掘漏洞	发现漏洞极其困难

在DREAD中每个因素都可分为高、中、低三个等级。在上表中每个等级分别以3、2、1的分数代表权重值，因此我们可以具体计算出某个风险的风险值。

安全世界观

信息安全三要素

安全问题的本质

> 安全问题的本质是“信任”问题。
>
> 我们可以通过"信任域的划分"、”信任边界的确定“来发现问题是在何处产生的。

一切的安全方案设计的基础都是建立在信任的关系上的。我们必须相信一些东西，必须有一些最基本的假设，安全方案才能得以建立，如果我们否定一切，安全方案就会如无源之水，无根之木，无法设计，也无法完成。

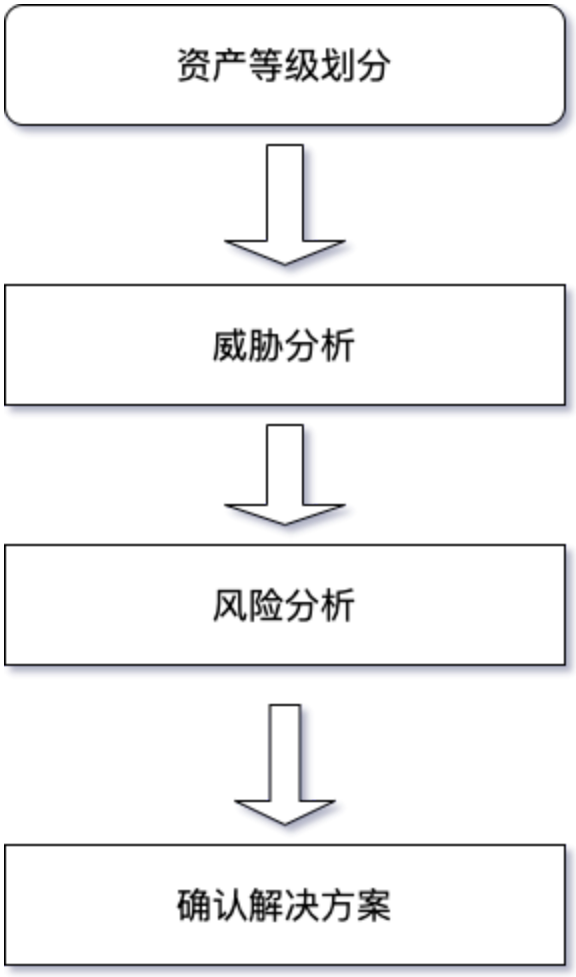
安全三要素

安全三要素是安全的基本组成元素，分别是机密性（Confidentiality）、完整性（Integrit）、可用性（Availability），简称CIA

> 机密性：要求保护数据内容不被泄漏，加密是实现机密性要求的常用手段。
>
> 完整性：要求保护的数据内容是完整的、没有被篡改的。常见的保证一致性的技术手段是数字签名。
>
> 可用性：要求保护的资源是“随需而得”，例：拒绝服务攻击（dos）,拒绝服务攻击破坏的就是安全的可用性，使系统无法提供正常服务。

安全评估

安全评估过程



资产等级划分

互联网安全的核心问题，是数据安全的问题。所以，对互联网公司的资产等级划分，就是对数据做等级划分，然后要划分信任域和信任边界。

威胁分析

威胁分析就是把所有的威胁都找出来。
一般采用头脑风暴法和威胁建模法（比如微软提出的STRIDE模型）。
漏洞的定义：系统中可能被威胁利用以造成危害的地方。
Here is yuque doc card, click on the link to view:https://www.yuque.com/u316337/kne2qg/rt1uyi

风险分析

风险由以下因素组成：Risk（风险）=Probability（发生的可能性）*Damage Potential（损失的大小）
一种科学衡量风险的方法：微软提出的DREAD模型。
注意：模型是死的，人是活的。模型只能起到辅助作用，人要灵活运用。

设计安全方案

一个优秀的安全方案应该具备以下特点：
能够有效解决问题
用户体验好
高性能
低耦合
易于扩展与升级

安全方案设计技巧

Secure By Default原则

- 黑名单、白名单原则（尽可能使用白名单，不用或者少用黑名单）
- 最小权限原则（要求系统只授予主体必要的权限，而不要过度授权）

Defence In Depth（纵深防御）原则

- 首先，要在各个不同层面、不同方面实施安全方案，避免出现疏漏，不同安全方案之间需要相互配合，构成一个整体；
- 其次，要在正确的地方做正确的事情。即：在解决根本问题的地方实施针对性的安全方案。

数据与代码分离原则

适用于各种由于注入而引发安全问题的场景，如xss，防止用户输入中包含代码。

不可预测性原则

- 不可预测性原则能够有效地对抗基于篡改、伪造的攻击。
- 不可预测的实现往往需要用的加密算法、随机数算法、哈希算法等，如解决csrf的token,攻击者在实施csrf攻击时是无法提前预知这个token的，只要token足够复杂就不会被攻击者猜测到；再就是如防爬虫的id hash,如用户访问/detail?id=1, id=2 这些纯数字的容易被猜到，可以变成 /detail?id=bhjBhfdbsvhejafekw，id=vjhdksbvhdjsbvfdj 这种形式。

浏览器安全

同源策略

- 同源策略是一种约定，它是浏览器最核心也最基本的安全功能。
- 浏览器的同源策略，限制了来自不同源的“document”或脚本，对当前“document”读取或者设置某些属性。
- 影响“源”的因素有：host、子域名、端口、协议。
- 在浏览器中，<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。
- 对于浏览器来说，除了DOM、Cookie、XMLHttpRequest会受到同源策略的限制外，浏览器加载的一些第三方插件也有各自的同源策略。最常见的一些插件如Flash、Java Applet、Silverlight、Google Gears等都有自己的控制策略。

浏览器沙箱

- 挂马：在网页中插入一段恶意代码，利用浏览器漏洞执行任意代码的攻击方式。
- 浏览器的多进程架构，将浏览器的各个功能模块分开，各个浏览器实例分开，当一个进程崩溃时，也不会影响到其他的进程。
- SandBox即沙箱，泛指“资源隔离类模块”，设计目的是为了 让不可信任的代码运行在一定的环境中，限制不可信任的代码访问隔离区之外的资源。

恶意网址拦截

- 工作原理：浏览器周期性地从服务端获取一份最新的恶意网址黑名单，如果用户上网时访问的网址存在于此黑名单中，浏览器就会弹出一个警告页面。
- 常见的恶意网址分为2类：挂马网站&钓鱼网站。
- 除了恶意网址黑名单拦截功能外，主流浏览器都开始支持EV SSL证书，以增强对安全网站的识别。

高速发展的浏览器安全

- 微软在IE8推出来XSS Filter功能，用以对抗反射型XSS。
- FireFox在FireFox4推出来Centent Security Policy（CSP），但并未得到推广。
- 浏览器加载的插件也是浏览器安全需要考虑的一个问题。

谈谈架构

架构：

架构的本质在于不断拆分生命周期（树形结构），使得业务可以做到空间上并行。拆出来的每一个生命周期都有自己的边界，不会影响到其他生命周期，各自的变化都在自己的生命周期内确定，即为高内聚。

软件生命周期： 软件开发生命周期 + 软件运行生命周期（软件访问、软件功能、软件监控）

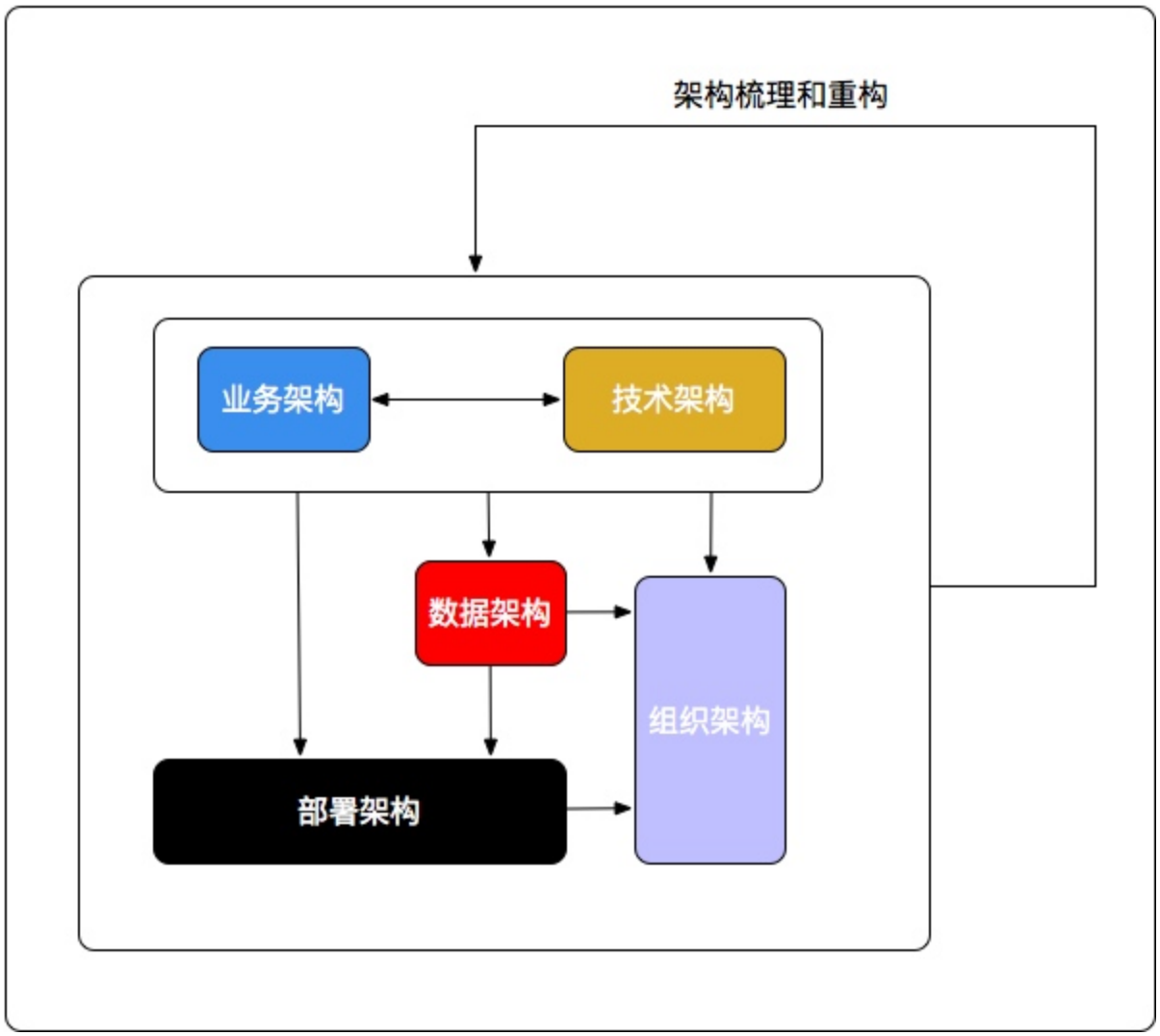
架构目标

高性能：提供快速的访问体验。
高可用：网站服务一直可以正常访问。
可伸缩：通过硬件增加/减少，提高/降低处理能力。
安全性：提供网站安全访问和数据加密，安全存储等策略。
扩展性：方便的通过新增/移除方式，增加/减少新的功能/模块。
敏捷性：随需应变，快速响应；

大型网站架构模式

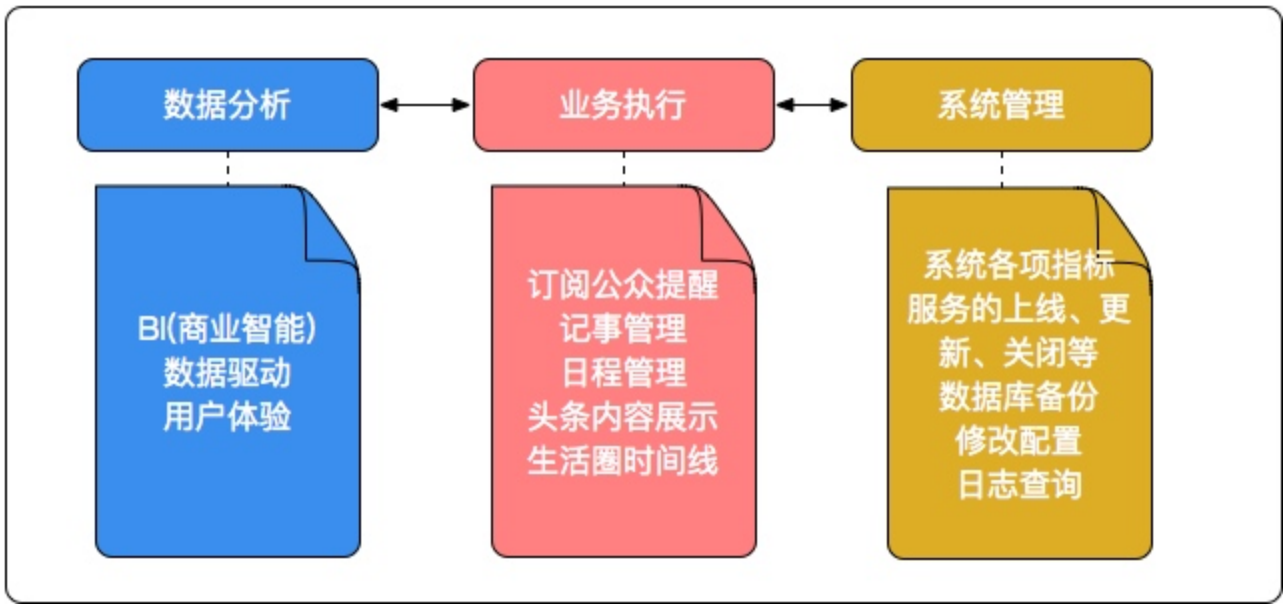
分层：一般可分为，应用层，服务层，数据层，管理层，分析层；
分割：一般按照业务/模块/功能特点进行划分，比如应用层分为首页，用户中心。
分布式：将应用分开部署（比如多台物理机），通过远程调用协同工作。
集群：一个应用/模块/功能部署多份（如：多台物理机），通过负载均衡共同提供对外访问。
缓存：将数据放在距离应用或用户最近的位置，加快访问速度。
异步：将同步的操作异步化。客户端发出请求，不等待服务端响应，等服务端处理完毕后，使用通知或轮询的方式告知请求方。一般指：请求——响应——通知 模式。
冗余：增加副本，提高可用性，安全性，性能。
安全：对已知问题有有效的解决方案，对未知/潜在问题建立发现和防御机制。
自动化：将重复的，不需要人工参与的事情，通过工具的方式，使用机器完成。
敏捷性：积极接受需求变更，快速响应业务发展需求。

架构流程概述



业务架构：

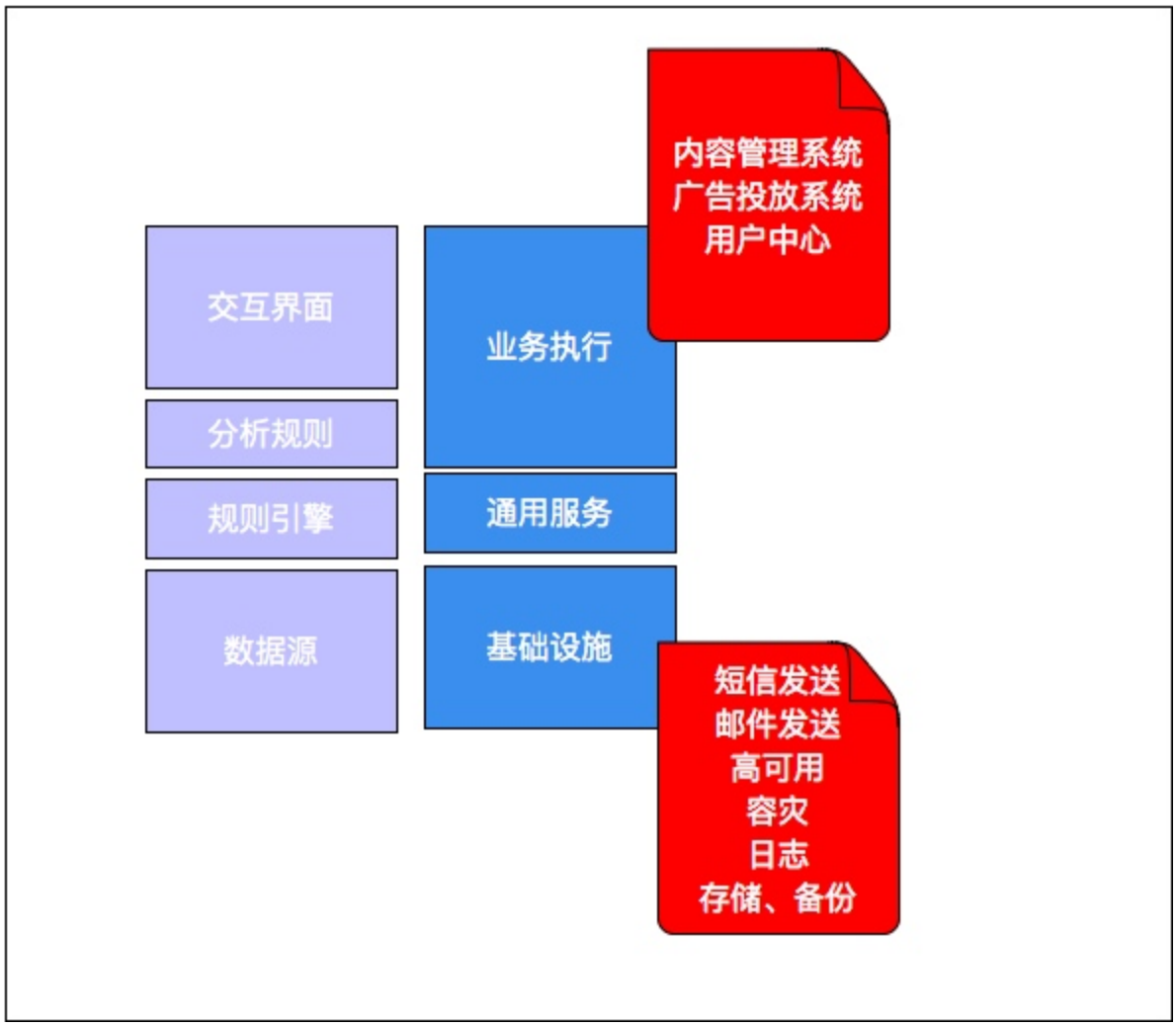
俯视架构，包括业务规则、业务模块和业务流程。主要是对整个系统的业务进行拆分，对领域模型进行设计，把现实中的业务转化成抽象的对象。



业务执行是应用的核心模块，是应用的主要功能。
数据分析是应用的辅助模块，有助于对应用做数据驱动化研发、商业智能研究、改善用户体验。
系统管理是应用的基础部分，做好系统的部署、各项指标的监控、关键数据的备份等有助于应用快速迭代部署和稳定运行。

技术架构：

剖面架构，是硬件到应用的抽象，包括抽象层和编程接口。技术架构和业务架构是相辅相成的关系，业务架构的每一个部分都有其技术架构，系统的架构需要先做好这两部分。



业务数据源、数据的规则引擎和分析规则支撑起了交互界面的呈现
基础设施、通用服务构建起了底层的业务逻辑

数据架构：

存储架构，主要指的是数据结构的设计。决定了应用数据源的特性，是业务架构和技术架构的基础。

交互界面呈现出来的数据交互逻辑、数据流向决定了业务的主要数据设计
原始的业务数据、日志和统计需要的数据支撑了数据分析需要的报表输出
实时信息在一定的规则和状态机引擎下可以提供出实时状态监测等仪表盘功能

数据的五个属性

访问频率：读写频率；只读且经常被访问的数据可以冗余多份
对一致性的要求：一致性要求高的数据需要严格保证准确性。
访问权限：API设计中根据不同的权限暴露不同粒度的数据。PO->VO即是对同一事物在不同权限下的描述，
数据重要性：不可丢失、允许部分丢失、只是缓存、无需保存
数据保密性：内部可以明文、内部不可明文、可以对外公开

不同性质的数据使用不同的数据架构策略， 数据统计场景中，实时性要求较高的数据统计可以用redis；非实时数据则可以使用单独表，通过队列异步运算或者定时计算更新数据。此外，对于一致性要求较高的统计数据，需要依靠事务或者定时校对机制保证准确性。

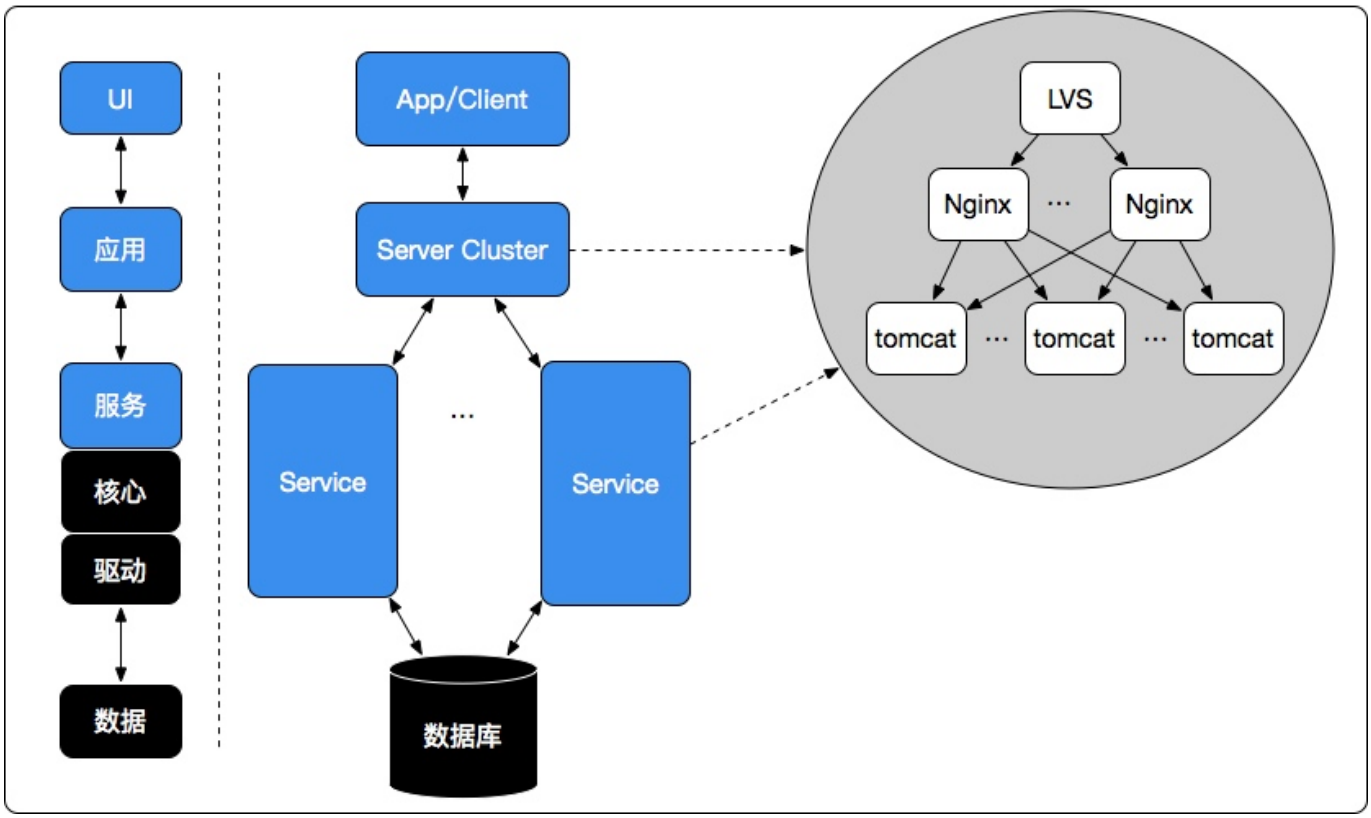
数据库设计需要注意存储效率：

减少事务
减少联表查询

适当使用索引
考虑使用缓存

部署架构：

拓扑架构，包括系统部署了几个结点、结点之间的关系、服务器的高可用、容错性、网络接口与协议等。决定了应用如何运行、运行的性能、可维护性、可扩展性等，是所有架构的基础。



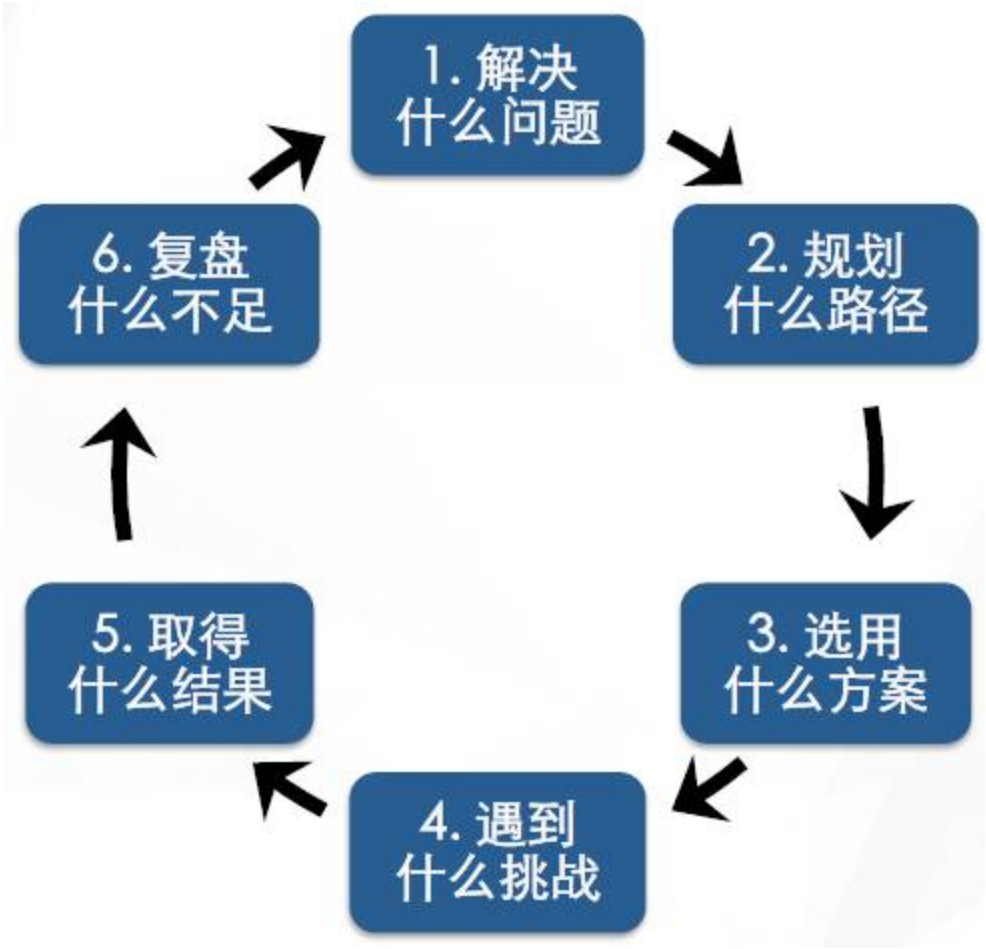
UI交互界面单独部署，包括web、app等形式
应用单独部署为一个结点或者集群
服务、核心以及驱动做为整体进行部署
数据源单独部署
一个简单的应用/服务集群: LVS(使用keepalived做主备) + Nginx(反向代理) + Tomcat(业务容器)

组织架构：

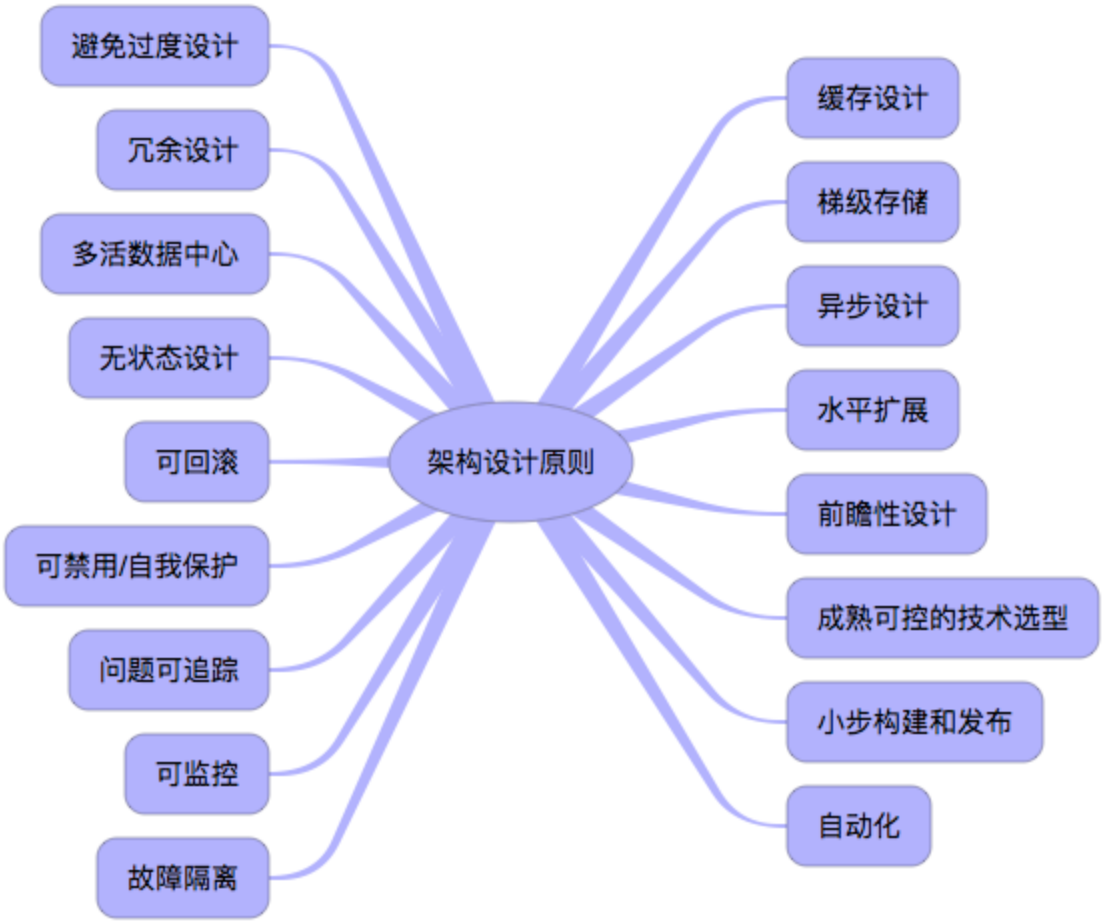
团队架构，包括项目的组织形式、人员构成、职责等，是上面所有架构的保障设施。良好的组织架构能够保证其他架构的有效实施和推进。

架构随着业务、负载的变化需要不断的梳理和重构，推进架构的演进。

架构六步思考法



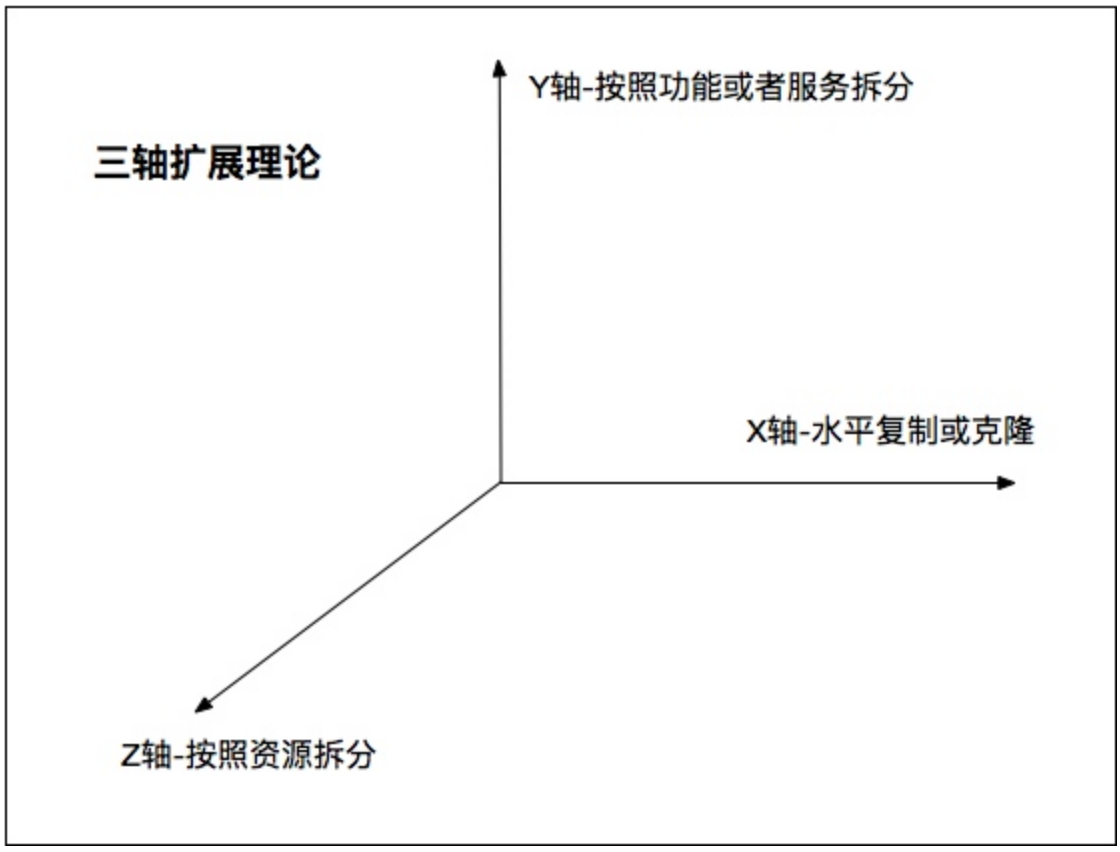
架构原则



避免过度设计：最简单的方案最容易实现和维护，也可以避免浪费资源。但方案中需要包括扩展。
冗余设计：对服务、数据库的做结点冗余，保证服务的高可用。通过数据库主从模式、应用集群来实现。
多活数据中心：为了容灾，从根本上保障应用的高可用性。需要构建多活的数据中心，以防止一个数据中心由于不可控因素出现故障后，引起整个系统的不可用。
无状态设计：api、接口等的设计不能有前后依赖关系，一个资源不受其他资源改动的影响。无状态的系统才能更好地进行扩展。如果非得有状态，则要么客户端管理状态，要么服务端用分布式缓存管理状态。
可回滚：对于任何业务尤其是关键业务，都具有恢复机制。可以使用基于日志的WAL、基于事件的Event sourcing等来实现可回滚。
可禁用/自我保护：具有限流机制，当上游的流量超过自身的负载能力时，能够拒绝溢出的请求。可以通过手动开关或者自动开关（监测异常流量行为），在应用前端挡住流量。此外，永远不要信赖第三方服务的可靠性，依赖于第三方的功能务必有服务降级措施以及熔断管理。
问题可追踪：当系统出现问题时，能够定位请求的轨迹、每一步的请求信息等。分布式链路追踪系统即解决的此方面的问题。
可监控：可监控是保障系统能够稳定运行的关键。包括对业务逻辑的监控、应用进程的监控以及应用依赖的CPU、硬盘等系统资源的监控。每一个系统都需要做好这几个层面的监控。
故障隔离：将系统依赖的资源(线程、CPU)和服务隔离开来能够使得某个服务的故障不会影响其他服务的调用。通过线程池或者分散部署结点可以对故障进行隔离。

成熟可控的技术选型：使用市面上主流、成熟、文档、支持资源多的技术，选择合适的而非最火的技术实现系统。
梯级存储：内存–>SSD硬盘–>传统硬盘–>磁带，可以根据数据的重要性和生命周期对数据进行分级存储。
缓存设计：隔离请求与后端逻辑、存储，是就近原则的一种机制。包括客户端缓存（预先下发资源）、nginx缓存、本地缓存以及分布式缓存。
异步设计：对于调用方不关注结果或者允许结果延时返回的接口，采用队列进行异步响应能够很大程度提高系统性能；调用其他服务的时候不去等待服务方返回结果直接返回，同样能够提升系统响应性能。异步队列也是解决分布式事务的常用手段。
前瞻性设计：根据行业经验和预判，提前把可扩展性、后向兼容性设计好。
水平扩展：相比起垂直扩展，能够通过堆机器解决问题是最优先考虑的问题，系统的负载能力也才能接近无限扩展。此外，基于云计算技术根据系统的负载自动调整容量能够在节省成本的同时保证服务的可用性。
小步构建和发布：快速迭代项目，快速试错。不能有跨度时间过长的项目规划。
自动化：打包、测试的自动化称为持续集成，部署的自动化称为持续部署。自动化机制是快速迭代和试错的基础保证。deployer

可扩展原则：



X轴，水平复制或克隆，面向目标，如数据库读写分离、复制表、replication等，将单体应用或者以来的服务做冗余，通过**负载均衡**提高系统负载能力。

Y轴，面向功能/服务，如垂直应用、分布式服务等，就是将单体应用根据**功能拆分**成小的应用或者服务。
Z轴，面向资源，如数据库**水平分库**，对资源做分片，将压力分散到不同的结点上。
避免依赖于数据库的运算功能(函数、存储器、触发器等)，将负载放在更容易扩展的业务应用端。

系统响应性能提升五板斧

异步：队列缓冲、异步请求。
并发：利用多CPU多线程执行业务逻辑。
就近原则：缓存、梯度存储。
减少IO：合并细粒度接口为粗粒度接口、频繁的覆盖操作可以只做最后一次操作。这里一个需要特别注意的地方: 代码中尽量避免在循环中调用外部服务，更好的做法是使用粗粒度批量接口在循环外面只进行一次请求。
分区：频繁访问的数据集规模保持在合理的范围。

架构示例

高性能架构

以用户为中心，提供快速的网页访问体验。主要参数有较短的响应时间，较大的并发处理能力，较高的吞吐量，稳定的性能参数。

可分为前端优化，应用层优化，代码层优化，存储层优化。

前端优化：网站业务逻辑之前的部分；

浏览器优化：减少Http请求数，使用浏览器缓存，启用压缩，Css Js位置，Js异步，减少Cookie传输；

CDN加速，反向代理；

应用层优化：处理网站业务的服务器。使用缓存，异步，集群

代码优化：合理的架构，多线程，资源复用（对象池，线程池等），良好的数据结构，JVM调优，单例，Cache等；

存储优化：缓存，固态硬盘，光纤传输，优化读写，磁盘冗余，分布式存储（HDFS），NOSQL等；

高可用架构

大型网站应该在任何时候都可以正常访问。正常提供对外服务。因为大型网站的复杂性，分布式，廉价服务器，开源数据库，操作系统等特点。要保证高可用是很困难的，也就是说网站的故障是不可避免的。

如何提高可用性，就是需要迫切解决的问题。首先，需要从架构级别，在规划的时候，就考虑可用性。行业内一般用几个9表示可用性指标。比如四个9（99.99），一年内允许的不可用时间是53分钟。

不同层级使用的策略不同，一般采用冗余备份和失效转移解决高可用问题。

应用层：一般设计为无状态的，对于每次请求，使用哪一台服务器处理是没有影响的。一般使用负载均衡技术（需要解决Session同步问题），实现高可用。

服务层：负载均衡，分级管理，快速失败（超时设置），异步调用，服务降级，幂等设计等。

数据层：冗余备份（冷，热备[同步，异步]，温备），失效转移（确认，转移，恢复）。数据高可用方面著名的理论基础是CAP理论（持久性，可用性，数据一致性[强一致，用户一致，最终一致]）

可伸缩架构

伸缩性是指在不改变原有架构设计的基础上，通过添加/减少硬件（服务器）的方式，提高/降低系统的处理能力。

应用层：对应用进行垂直或水平切分。然后针对单一功能进行负载均衡（DNS,HTTP[反向代理],IP,链路层）。

服务层：与应用层类似；

数据层：分库，分表，NOSQL等；常用算法Hash，一致性Hash。

可扩展架构

可以方便的进行功能模块的新增/移除，提供代码/模块级别良好的可扩展性。

模块化，组件化：高内聚，低耦合，提高复用性，扩展性。

稳定接口：定义稳定的接口，在接口不变的情况下，内部结构可以“随意”变化。

设计模式：应用面向对象思想，原则，使用设计模式，进行代码层面的设计。

消息队列：模块化的系统，通过消息队列进行交互，使模块之间的依赖解耦。

分布式服务：公用模块服务化，提供其他系统使用，提高可重用性，扩展性。

安全架构

对已知问题有有效的解决方案，对未知/潜在问题建立发现和防御机制。对于安全问题，首先要提高安全意识，建立一个安全的有效机制，从政策层面，组织层面进行保障。比如服务器密码不能泄露，密码每月更新，并且三次内不能重复；每周安全扫描等。以制度化的方式，加强安全体系的建设。同时，需要注意与安全有关的各个环节。安全问题不容忽视。包括基础设施安全，应用系统安全，数据保密安全等。

基础设施安全：硬件采购，操作系统，网络环境方面的安全。一般采用，正规渠道购买高质量的产品，选择安全的操作系统，及时修补漏洞，安装杀毒软件防火墙。防范病毒，后门。设置**防火墙**策略，建立**DDOS防御**系统，使用攻击检测系统，进行子网隔离等手段。

应用系统安全：在程序开发时，对已知常用问题，使用正确的方式，在代码层面解决掉。防止**跨站脚本攻击（XSS），注入攻击，跨站请求伪造（CSRF），错误信息，HTML注释，文件上传，路径遍历**等。还可以使用Web应用防火墙（比如：ModSecurity），进行安全漏洞扫描等措施，加强应用级别的安全。

数据保密安全：存储安全（存在在可靠的设备，实时，定时备份），保存安全（重要的信息加密保存，选择合适的人员复杂保存和检测等），传输安全（防止数据窃取和数据篡改）；

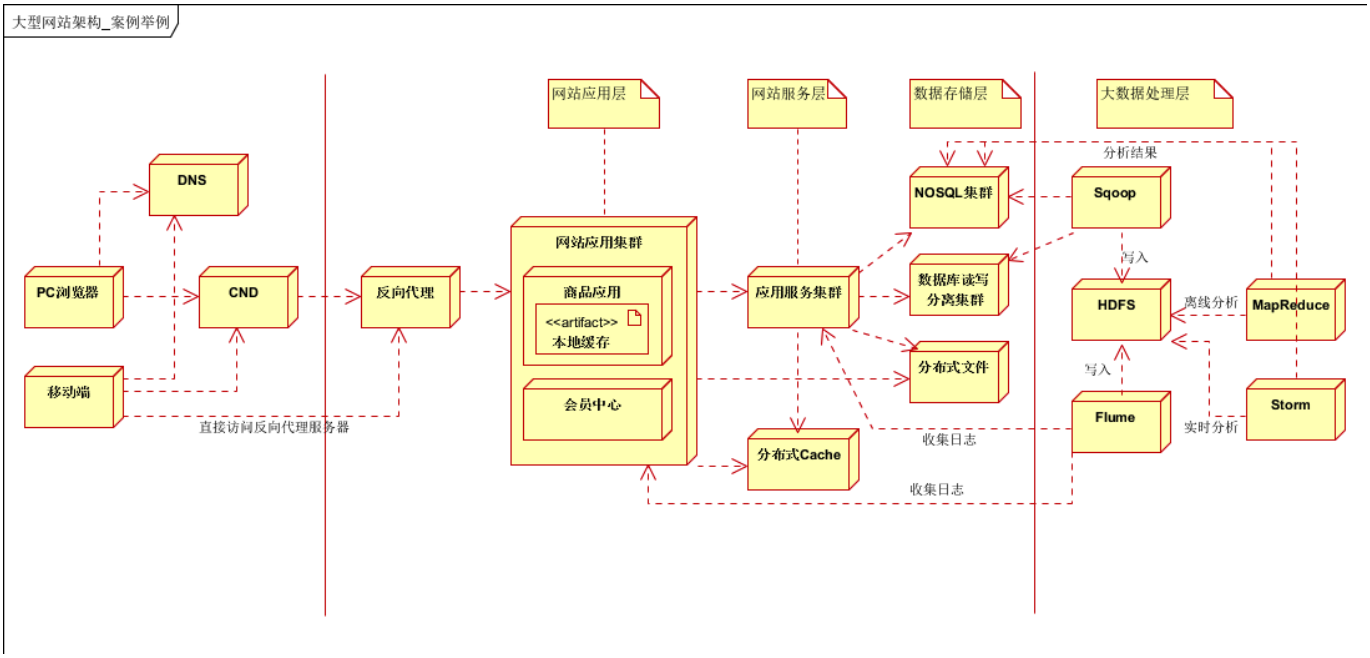
常用的加解密算法（单项散列加密[MD5,SHA]，对称加密[DES,3DES,RC]），非对称加密[RSA]等。

敏捷性

网站的架构设计，运维管理要适应变化，提供高伸缩性，高扩展性。方便的应对快速的业务发展，突增高流量访问等要求。

除上面介绍的架构要素外，还需要引入敏捷管理，敏捷开发的思想。使业务，产品，技术，运维统一起来，随需应变，快速响应。

大型架构举例



以上采用七层逻辑架构，第一层客户层，第二层前端优化层，第三层应用层，第四层服务层，第五层数据存储层，第六层大数据存储层，第七层大数据处理层。

客户层：支持PC浏览器和手机APP。差别是手机APP可以直接访问通过IP访问，反向代理服务器。

前端层：使用DNS负载均衡，CDN本地加速以及反向代理服务；

应用层：网站应用集群；按照业务进行垂直拆分，比如商品应用，会员中心等；

服务层：提供公用服务，比如用户服务，订单服务，支付服务等；

数据层：支持关系型数据库集群（支持读写分离），NOSQL集群，分布式文件系统集群；以及分布式Cache；

大数据存储层：支持应用层和服务层的日志数据收集，关系数据库和NOSQL数据库的结构化和半结构化数据收集；

大数据处理层：通过Mapreduce进行离线数据分析或Storm实时数据分析，并将处理后的数据存入关系型数据库。（实际使用中，离线数据和实时数据会按照业务要求进行分类处理，并存入不同的数据库中，供应用层或服务层使用）。

系统负载介绍

负载

负载就是cpu在一段时间内正在处理以及等待cpu处理的进程数之和的统计信息，也就是cpu使用队列的长度统计信息，这个数字越小越好（如果超过CPU核心*0.7就是不正常）

负载分为两大部分：CPU负载、IO负载

例如，假设有一个进行大规模科学计算的程序，虽然该程序不会频繁地从磁盘输入输出，但是处理完成需要相当长的时间。因为该程序主要被用来做计算、逻辑判断等处理，所以程序的处理速度主要依赖于cpu的计算速度。此类cpu负载的程序称为“计算密集型程序”。

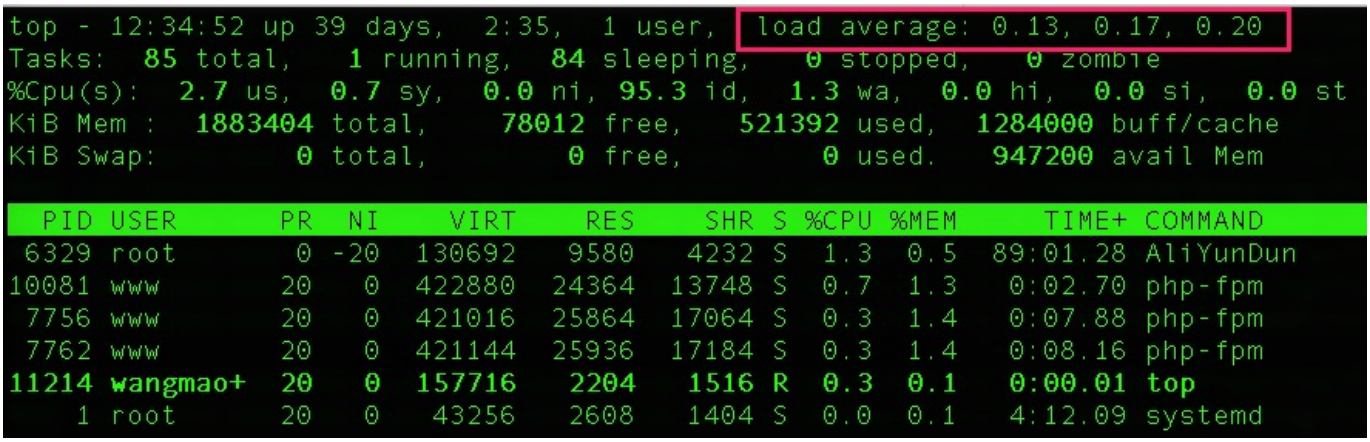
还有一类程序，主要从磁盘保存的大量数据中搜索找出任意文件。这个搜索程序的处理速度并不依赖于cpu，而是依赖于磁盘的读取速度，也就是输入输出（input/output,I/O）.磁盘越快，检索花费的时间就越短。此类I/O负载的程序，称为“I/O密集型程序”。

多任务操作系统

Linux操作系统能够同时处理几个不同名称的任务。但是同时运行多个任务的过程中，cpu和磁盘这些有限的硬件资源就需要被这些任务程序共享。即便很短的时间间隔内，需要一边在这些任务之间进行切换到一边进行处理，这就是多任务。

运行中的任务较少的情况下，系统并不是等待此类切换动作的发生。但是当任务增加时，例如任务A正在CPU上执行计算，接下来如果任务B和C也想进行计算，那么就需要等待CPU空闲。也就是说，即便是运行处理某任务，也要等到轮到他时才能运行，此类等待状态就表现为程序运行延迟。

top输出中包含“load average”



Load average从左边起依次是过去1分钟、5分钟、15分钟内，单位时间的等待任务数，也就是表示平均有多少任务正处于等待状态。在load average较高的情况下，这就说明等待运行的任务较多，因此轮到该任务运行的等待时间就会出现较大的延迟，即反映了此时负载较高。

进程调度

进程调度也被一些人称为cpu上下文切换意思是：CPU切换到另一个进程需要保存当前进程的状态并恢复另一个进程的状态：当前运行任务转为就绪（或者挂起、中断）状态，另一个被选定的就绪任务成为当前任务。进程调度包括保存当前任务的运行环境，恢复将要运行任务的运行环境。

在linux内核中，每一个进程都存在一个名为“进程描述符”的管理表。该进程描述符会调整为按照优先级降序排序，已按合理的顺序运行进程（任务）。这个调整即为进程调度器的工作。

调度器划分并管理进程的状态，如：

等待分配cpu资源的状态。
等待磁盘输入输出完毕的状态。
下面在说一下进程的状态区别：

状态	说明
运行态（running）	只要cpu空闲，任何时候都可以运行
可中断睡眠（interruptible）	为恢复时间无法预测的长时间等待状态。如，来自于键盘设备的输入。
不可中断睡眠:（uninterruptible）	主要为短时间时的等待状态。例如磁盘输入输出等待。被IO阻塞的进程
就绪态（runnable）	响应暂停信号而运行的中断状态。
僵死态（zombie）	进程都是由父进程创建，并销毁；在父进程没有销毁其子进程，被销毁的时候，其子进程由于没有父进程被销毁，就会转变为僵死态。

下面举例来说明进程状态转变：

这里有三个进程A、B、C同时运行。首先，每个进程在生成后都是可运行状态，也就是running状态的开始，而不是现在运行状态，由于在linux内核中无法区别正在运行的状态和可运行的等待状态，下面将可运行状态和正在运行状态都称为running状态。

进程A：running
进程B：running
进程C：running
running的三个进程立即成为调度对象。此时，假设调度器给进程A分配了CPU的运行权限。

进程A：running （正在运行）
进程B：running
进程C：running
进程A分配了CPU，所以进程A开始处理。进程B和C则在此等待进程A迁出CPU。假设进程A进行若干计算之后，需要从磁盘读取数据。那么在A发出读取磁盘数据的请求之后，到请求数据到达之前，将不进行任何工作。此状态称为“因等待I/O操作结束而被阻塞”。在I/O完成处理前，进程A就一直处于等待中，就会转为不可中断睡眠状态（uninterruptible），并不使用CPU。于是调度器查看进程B和进程C的优先级计算结果，将CPU运行权限交给优先级较高的一方。这里假设进程B的优先级高于进程C。

进程A：uninterruptible （等待磁盘输入输出/不可中断状态）
进程B：running （正在运行）
进程C：running
进程B刚开始运行，就需要等待用户的键盘输入。于是B进入等待用户键盘输入状态，同样被阻塞。结果就变成了进程A和进程B都是等待输出，运行进程C。这时进程A和进程B都是等待状态，但是等待磁盘输入输出和等待键盘输入为不同的状态。等待键盘输入是无限期的事件等待，而读取磁盘则是必须短时间内完成的事件等待，这是两种不同的等待状态。各进程状态如下所示：

进程A：uninterruptible （等待磁盘输入输出/不可中断状态）
进程B：interruptible （等待键盘输入输出/可中断状态）
进程C：running （正在运行）
这次假设进程C在运行的过程中，进程A请求的数据从磁盘到达了缓冲装置。紧接着硬盘对内核发起中断信号，内核知道磁盘读取完成，将进程A恢复为可运行状态。

进程A：running （正在运行）
进程B：interruptible （等待键盘输入输出/可中断状态）
进程C：running （正在运行）
此后进程C也会变为某种等待状态。如CPU的占用时间超出了上限、任务结束、进入I/O等待。一旦满足这些条件，调度器就可以完成从进程C到进程A的进程状态切换。

负载的意义：

负载表示的是“等待进程的平均数”。在上面的进程状态变换过程中，除了running状态，其他都是等待状态，那么其他状态都会加入到负载等待进程中吗?

事实证明，只有进程处于运行态（running）和不可中断状态（interruptible）才会被加入到负载等待进程中，也就是下面这两种情况的进程才会表现为负载的值。

即便需要立即使用CPU，也还需等待其他进程用完CPU
即便需要继续处理，也必须等待磁盘输入输出完成才能进行

下面描述一种直观感受的场景说明为什么只有运行态（running）和可中断状态（interruptible）才会被加入负载。

如：在很占用CPU资源的处理中，例如在进行动画编码的过程中，虽然想进行其他相同类型的处理，结果系统反映却变得很慢，还有从磁盘读取大量数据时，系统的反映也同样会变的很慢。但是另一方面，无论有多少等待键盘输入输出操作的进程，也不会让系统响应变慢。

造成CPU低而负载高场景

通过上面的具体分析负载的意义就很明显了，负载总结为一句话就是：需要运行处理但又必须等待队列前的进程处理完成的进程个数。具体来说，也就是如下两种情况：

等待被授权予CPU运行权限的进程
等待磁盘I/O完成的进程
cpu低而负载高也就是说等待磁盘I/O完成的进程过多，就会导致队列长度过大，这样就体现到负载过大了，但实际是此时cpu被分配去执行别的任务或空闲，具体场景有如下几种。

场景一：磁盘读写请求过多就会导致大量I/O等待

上面说过，cpu的工作效率要高于磁盘，而进程在cpu上面运行需要访问磁盘文件，这个时候cpu会向内核发起调用文件的请求，让内核去磁盘取文件，这个时候会切换到其他进程或者空闲，这个任务就会转换为不可中断睡眠状态。当这种读写请求过多就会导致不可中断睡眠状态的进程过多，从而导致负载高，cpu低的情况。

场景二：MySQL中存在没有索引的语句或存在死锁等情况

我们都知道MySQL的数据是存储在硬盘中，如果需要进行sql查询，需要先把数据从磁盘加载到内存中。当在数据特别大的时候，如果执行的sql语句没有索引，就会造成扫描表的行数过大导致I/O阻塞，或者是语句中存在死锁，也会造成I/O阻塞，从而导致不可中断睡眠进程过多，导致负载过大。

具体解决方法可以在MySQL中运行show full processlist命令查看线程等待情况，把其中的语句拿出来进行优化。

场景三：外接硬盘故障，常见有挂了NFS，但是NFS server故障

比如我们的系统挂载了外接硬盘如NFS共享存储，经常会有大量的读写请求去访问NFS存储的文件，如果这个时候NFS Server故障，那么就会导致进程读写请求一直获取不到资源，从而进程一直是不可中断状态，造成负载很高。