1. Location语法

1. 语法规则

```
location [=|~|~*|^~] /uri/
{
    …
}
正则表达式中的特殊字符:
- . () {} [] * + ?
```

2. Location优先级说明
   - 在nginx的location和配置中location的顺序没有太大关系。
   - 与location表达式的类型有关。
   - **相同类型的表达式，字符串长的会优先匹配。**
3. location表达式类型
   - ~ 表示执行一个正则匹配，区分大小写
   - ~* 表示执行一个正则匹配，不区分大小写
   - ^~ 表示普通字符匹配。使用前缀匹配。如果匹配成功，则不再匹配其他location。
   - = 进行普通字符精确匹配。也就是完全匹配。
4. 匹配模式及优先级顺序(高 -> 低):

location = /uri	**=开头表示精确匹配，只有完全匹配上才能生效。 一旦匹配成功，则不再查找其他匹配项。**
location ^~ /uri	^~ 开头对URL路径进行前缀匹配，并且在正则之前。 一旦匹配成功，则不再查找其他匹配项。 需要考虑先后顺序, 如: http://localhost/helloworld/test/a.html
location ~ pattern location ~* pattern	~ 开头表示区分大小写的正则匹配。 ~*开头表示不区分大小写的正则匹配。
location /uri	不带任何修饰符，也表示前缀匹配，但是在正则匹配之后。
location /	通用匹配，任何未匹配到其它location的请求都会匹配到，相当于switch中的default。

```

客户端: http://localhost/helloworld/test/a.html
客户端: http://localhost/helloworld/test/
/helloworld/test/a.html
/helloworld/test/

location /
{
}
location /helloworld/
{
}
```

```
location /helloworld/test/
{
}


location =/helloworld/test/
{
    root xxx;
}


http://localhost/helloworld/test/a.html
location ^~ /helloworld/test/
{
}


location ^~ /login/
{
}
http://localhost/helloworld/test/a.JPG
location ~* \.[jpg|png]
{
}

http://192.168.1.100/login/hello/world/login.html
/login/hello/world/login.html
location /
{
}
location /login/
{
}
location /login/hello/
{
}
location /login/hello/world/
{
}
location ~ /group[1-9]/M0[0-9]
{
}
```

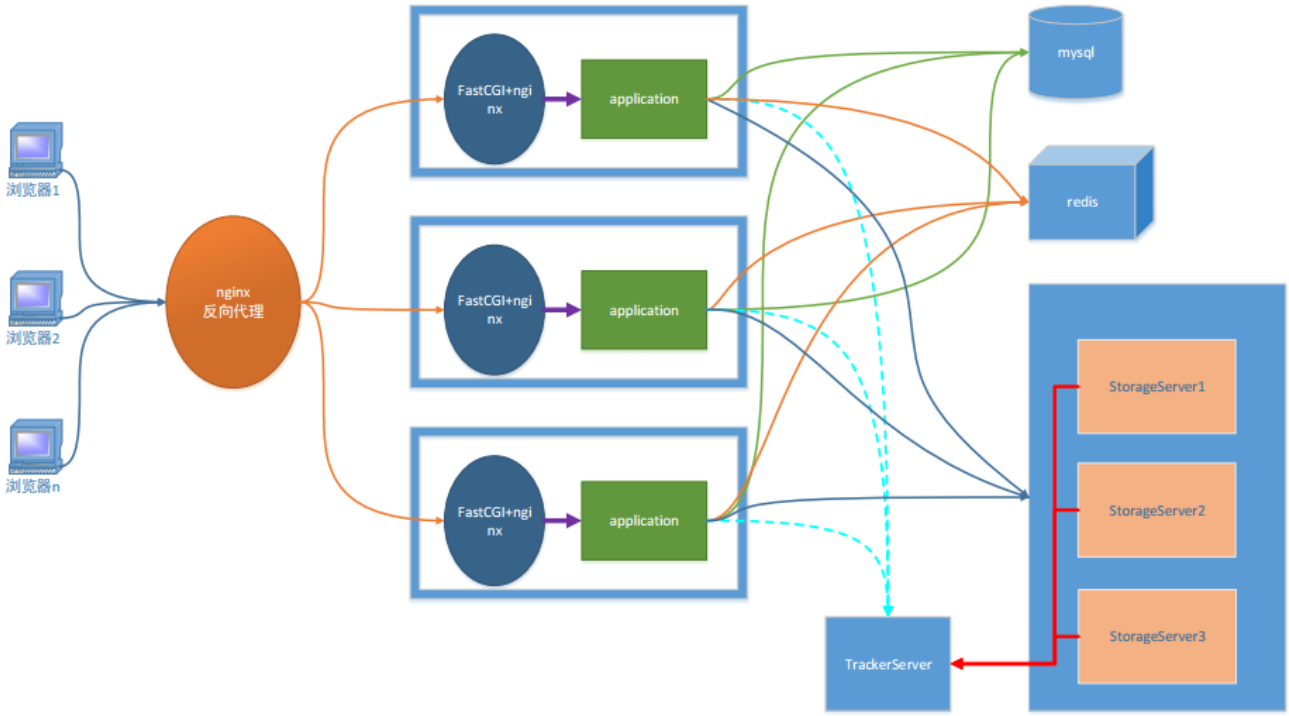
5. 练习题

```
location = / {
    # 仅仅匹配请求 /
    [ configuration A ]
}
location / {
    # 匹配所有以 / 开头的请求。
    # 但是如果有更长的同类型的表达式，则选择更长的表达式。
    # 如果有正则表达式可以匹配，则优先匹配正则表达式。
    [ configuration B ]
}
location /documents/ {
    # 匹配所有以 /documents/ 开头的请求。
    # 但是如果有更长的同类型的表达式，则选择更长的表达式。
    # 如果有正则表达式可以匹配，则优先匹配正则表达式。
    [ configuration C ]
}
location ^~ /images/ {
    # 匹配所有以 /images/ 开头的表达式，如果匹配成功，则停止匹配查找。
    # 所以，即便有符合的正则表达式location，也不会被使用
    [ configuration D ]
}
location ~* \.(gif|jpg|jpeg)$ {
    # 匹配所有以 gif jpg jpeg结尾的请求。
    # 但是 以 /images/开头的请求，将使用 Configuration D
    [ configuration E ]
}
```

匹配示例:

- / -> configuration
- /index.html -> configuration
- /documents/document.html -> configuration C
  - /documents/
  - /
- /images/1.gif -> configuration D
  - /images/
  - /
- /documents/1.jpg -> configuration E

2. 项目总结



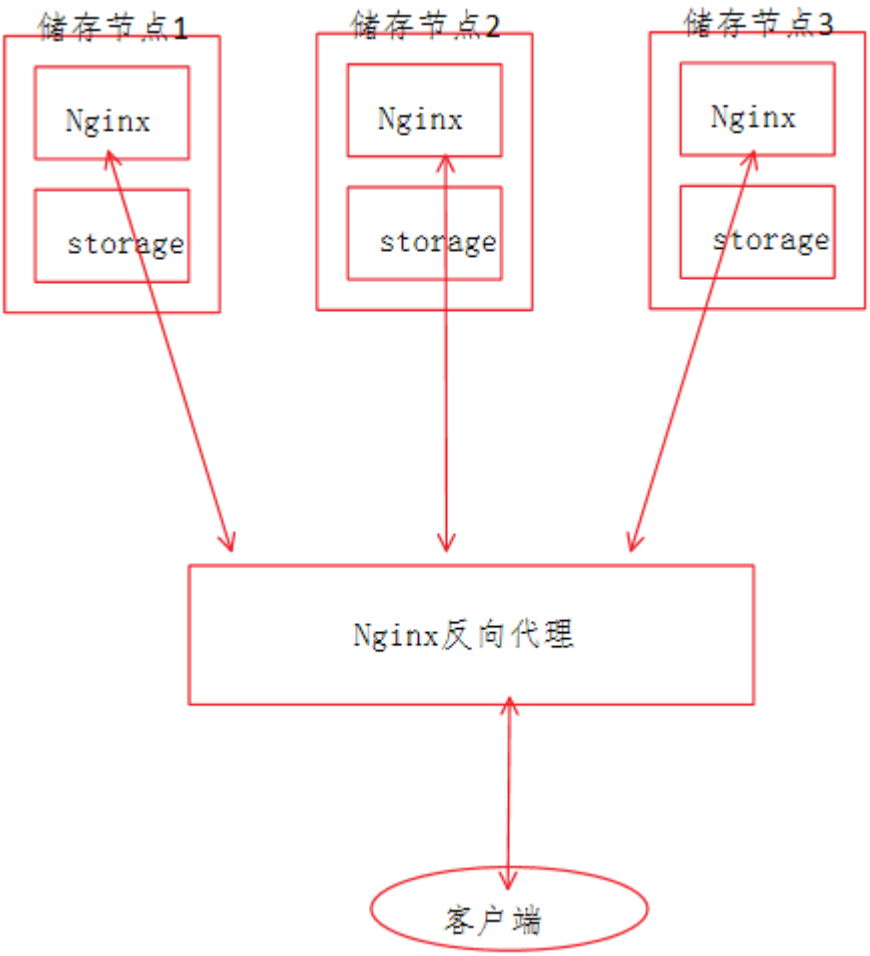
1. 客户端
    - Qt
        - 了解了Qt中http通信
2. nginx反向代理服务器
    - 为web服务器服务
    - web服务器需要集群
3. web服务器 - nginx
    - 处理静态请求 - > 访问服务器文件
    - 动态请求 -> 客户端给服务器提交的数据
        - 借助fastCGI进行处理
            - 讲的是单线程处理方式 - API
            - 也可以多线程处理 -> 另外的API
            - 使用spawn-fcgi启动
4. mysql
    - 关系型数据库 - 服务器端
    - 存储什么?
        - 项目中所有用到的数据
5. redis
    - 非关系型数据库 - 服务器端使用
    - 数据默认在内存, 不需要sql语句, 不需要数据库表
    - 键值对存储, 操作数据使用的是命令
    - 和关系型数据库配合使用
    - 存储服务器端经常访问的数据

6. fastDFS
    - 分布式文件系统
    - 追踪器, 存储节点, 客户端
    - 存储节点的集群
        - 横向扩容 -> 增加容量
            - 添加新组, 将新主机放到该组中
        - 纵向扩容 -> 备份
            - 将主机放到已经存在的组中
        - 存储用户上传的所有的文件
        - 给用户提供下载服务器

3. 项目提炼

1. 做的是文件服务器
    - 电商网站
    - 旅游网站
    - 租房
    - 装修公司
    - 医院
    - 短视频网站
2. 需要什么?
    - 首先需要的是fastDFS
        - 配置环境
        - 扩容
    - 操作fastDFS - 客户端
        - web
        - 桌面终端 - Qt
    - 数据库操作
        - mysql
        - oralce
    - 有一个web服务器 - Nginx
        - 静态资源部署
        - 动态请求 - 编写fastCGI程序
            - 注册
            - 登录
            - 上传
            - 下载
            - 秒传
            - 文件列表的获取
    - redis
        - 存储服务器访问比较频繁的数据

4. 存储节点反向代理



> http://192.168.31.109:80/group1/M00/00/00/wKgfbViy2Z2AJ-FTAaM3As g3Z0782.mp4"
>
> http://192.168.31.109:80/group2/M00/00/00/wKgfbViy2Z2AJ-FTAaM3As g3Z0782.mp4"

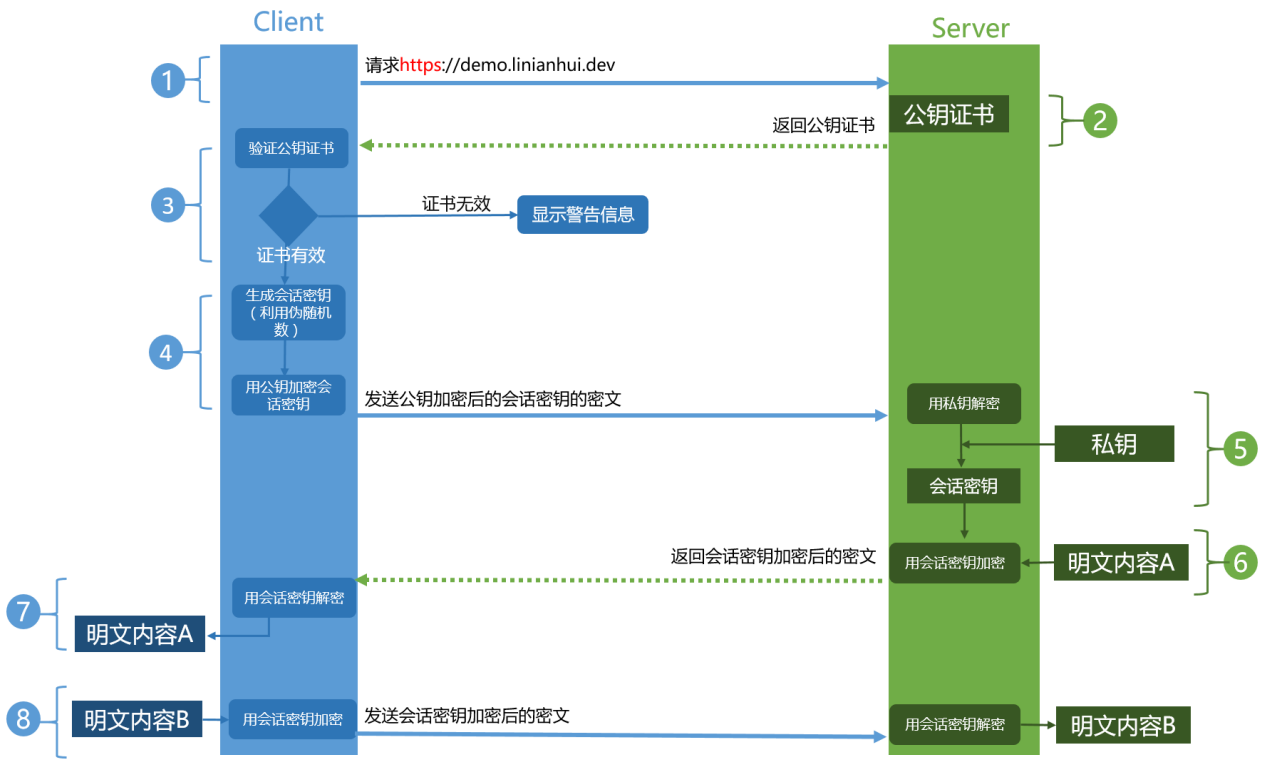
```
上图的反向代理服务器代理的是每个存储节点上部署的Nginx
    - 每个存储节点上的Nginx的职责：解析用户的http请求，帮助用户快速下载文件
客户端上传了一个文件，被存储到了fastDFS上，得到一个文件ID
    /group1/M00/00/00/wKgfbViy2Z2AJ-FTAaM3Asg3Z0782.mp4"
因为存储节点有若干个，所有下载的时候不知道对应的存储节点的访问地址
给存储节点上的nginx web服务器添加反向代理服务器之后，下载的访问地址：
    - 只需要知道nginx反向代理服务器的地址就可以了：192.168.31.109
    - 访问的url：
       http://192.168.31.109/group1/M00/00/00/wKgfbViy2Z2AJ-FTAaM3Asg3Z0782.mp4
客户端的请求发送给了nginx反向代理服务器
    - 反向代理服务器不处理请求，只转发，转发给存储节点上的nginx服务器
反向代理服务器的配置 - nginx.conf
    - 找出处理指令：去掉协议，iP/域名，末尾文件名，?和后边的字符串
        - /group1/M00/00/00/ - 完整的处理指令
    - 添加location
server{
    location  /group1/M00
```

```
        {
            # 数据转发，设置转发地址
            proxy_pass http://test.com;
        }
        location  /group2/M00
        {
            # 数据转发，设置转发地址
            proxy_pass http://test1.com;
        }
    }
    upstream test.com
    {
        # fastDFS存储节点的地址，因为存储节点上安装了nginx，安装的nginx作为web服务器的角色
        server 192.168.31.100;
        server 192.168.31.101;
        server 192.168.31.102;
    }
    upstream test1.com
    {
        # fastDFS存储节点的地址，因为存储节点上安装了nginx，安装的nginx作为web服务器的角色
        server 192.168.32.100;
        server 192.168.33.101;
        server 192.168.34.102;
    }

    # ====================================
    存储节点上的web服务器的配置
    存储节点1
        location  /group1/M00
        {
            # 请求处理
            root 请求的资源的根目录; // 存储节点的store_path0对应的路径+data
            ngx_fastdfs_module;
        }
        location  /group1/M01
        {
            # 请求处理
            root 请求的资源的根目录;
            ngx_fastdfs_module;
        }
    存储节点2
        location  /group2/M00
        {
            # 请求处理
            root 请求的资源的根目录;
            ngx_fastdfs_module;
        }
        location  /group2/M01
        {
            # 请求处理
            root 请求的资源的根目录;
            ngx_fastdfs_module;
        }
```

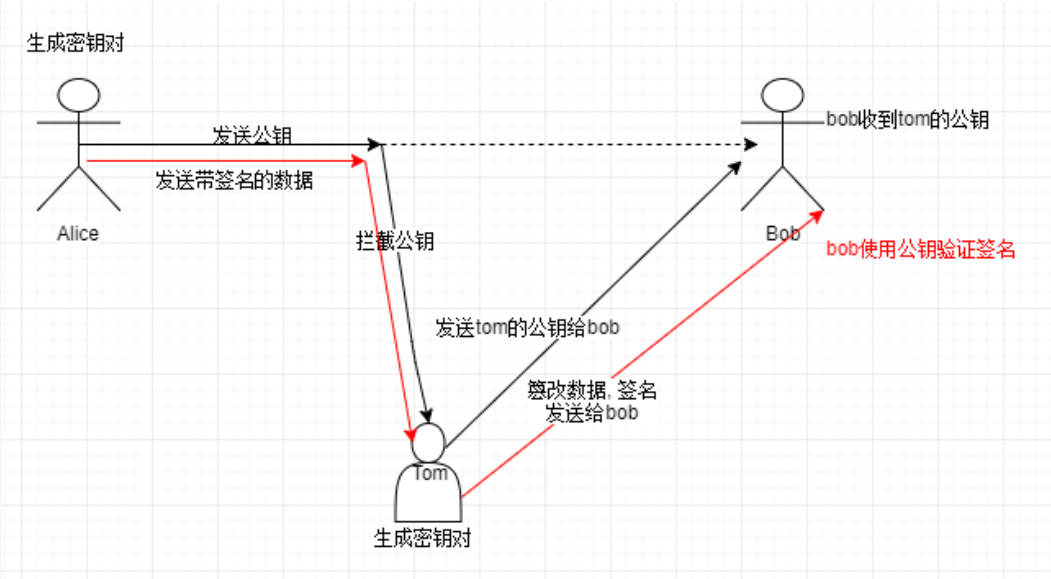
```
    存储节点3
        location  /group3/M00
        {
            # 请求处理
            root 请求的资源的根目录;
            ngx_fastdfs_module;
        }
        location  /group3/M01
        {
            # 请求处理
            root 请求的资源的根目录;
            ngx_fastdfs_module;
        }
```

5. https



1. 在百度服务器端首先生成一个秘钥对 -> 对公钥分发
2. 百度将公钥给到了CA认证机构, ca对私钥进行签名 -> 生成了证书.
3. 第一步第二部只做一次
4. 客户端访问百度, 百度将ca生成的证书发送给客户端
5. 浏览器对收到的证书进行认证
6. 如果证书没有问题 -> 使用ca的公钥将服务器的公钥从证书中取出
7. 我们得到了百度的公钥
8. 在浏览器端生成一个随机数, 使用得到的公钥进行加密, 发送给服务器
9. 服务器端使用私钥解密, 得到了随机数, 这个随机数就是对称加密的秘钥
10. 现在秘钥分发已经完成, 后边的通信使用的的对称加密的方式

1. 对称加密
    - 加解密秘钥是同一个
2. 非对称加密
    - 公钥, 私钥
        - rsa -> 公钥私钥都是两个数字
        - ecc -> 椭圆曲线, 两个点
    - 公钥加密, 私钥解密
        - 数据传输的时候使用
    - 私钥加密, 公钥解密
        - 数字签名
3. 哈希函数
    - md5/ sha1/sha2
    - 得到散列值, 散列值是定长的
4. 消息认证码
    - 生成消息认证码: (将原始数据+共享秘钥) * 进行哈希运算 = 散列值
    - 验证消息认证码:
        - (接收的原始数据 + 共享秘钥) * 哈希运算 = 新的散列值
        - 新散列值和旧散列值进行比较, 看是不是相同
    - 作用:
        - 验证数据的一致性型
    - 弊端:
        - 两端共享秘钥必须相同, 共享秘钥分发困难
5. 数字签名 -> 目的告诉所有人这个数据的所有者是xxx, xxx就是拿私钥的人
    - 生成一个非对称加密的密钥对
        - 公钥
        - 私钥
    - 生成签名:
        - 对原始数据进行哈希运算 -> 散列值
        - 使用非对称加密的私钥, 对散列值进行签名(私钥加密) -> 密文
        - 得到的密文就是数字签名
    - 签名的校验:
        - 校验这会收到签名者发送的数据
            - 原始数据
            - 数字签名
        - 对接收的数据进行哈希运算 -> 散列值
        - 使用非对称加密的公钥, 对数字签名进行解密 -> 明文 == 签名者生成的散列值
        - 校验者的散列值 和 签名者的散列值进行比较
            - 相同 -> 校验成功了, 数据属于签名的人
            - 失败 -> 数据不属于签名的人
    - 弊端:
        - 接收公钥的人没有办法校验公钥的所有者



6. 证书
    - 由一个受信赖的机构 (CA) 对某人的公钥进行数字签名
        - CA有一个密钥对
            - 使用ca的私钥对某个人的公钥进行加密 -> 证书
                - 这个人 的公钥
                - 这个人的个人信息