# MySQL

## 索引

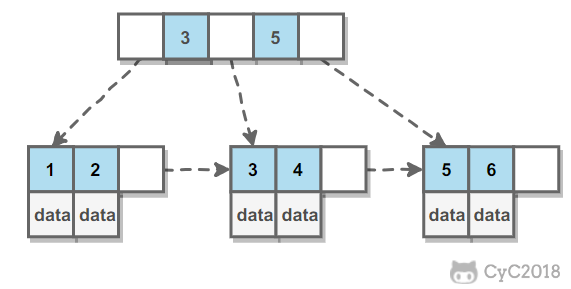
### B+Tree原理

#### 数据结构

B Tree指的是Balance Tree，也就是平衡树。平衡树是一颗查找树，并且所有的叶子节点都位于同一层。

B+Tree是基于B Tree和叶子节点顺序访问指针进行实现，它具有B Tree的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在B+Tree中，一个点中的Key从左到右非递减排列，如果某个指针的左右相邻key分别是和，且不为null，则该指针指向节点的所有key大于等于且小于等于。



#### 操作

进行查找操作时，首先在根节点进行二分查找，找到一个key所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出key所对应的data。

插入删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

#### 与红黑树的比较

红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用B+ Tree作为索引结构，主要有以下两个原因：

* 更少的查找次数：平衡树查找操作的时间复杂度和树高h相关，，其中d为每个节点的出度。红黑树的出度为2，而B+ Tree的出度一般都非常大，所以红黑树的树高h很明显比B+ Tree大非常多，查找的次数也就更多。
* 利用磁盘预读的特性：为了减少磁盘I/O操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的磁盘旋转时间，速度会非常快。操作系统一般将内存和磁盘分割成固定大小的块，每一块成为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次I/O就能完全载入一个节点。并且可以利用预读特性，相邻的节点也能够被预先载入。

### MySQL索引

索引是在存储引擎层实现的，而不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现。

#### B+Tree索引

B+Tree是大多数MySQL存储引擎的默认索引类型。

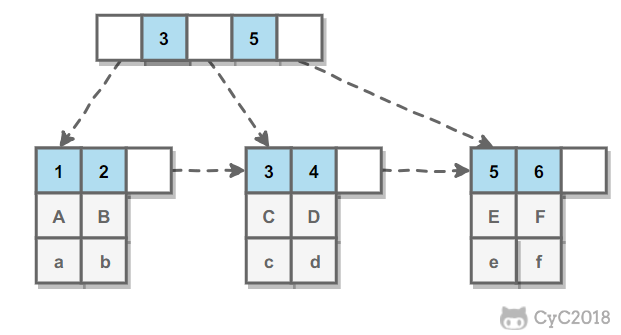
因为不再需要进行全表扫描，只需要对树进行搜索即可，所以查找速度快很多。

因为B+ Tree的有序性，所以除了用于查找，还可以用于排序和分组。

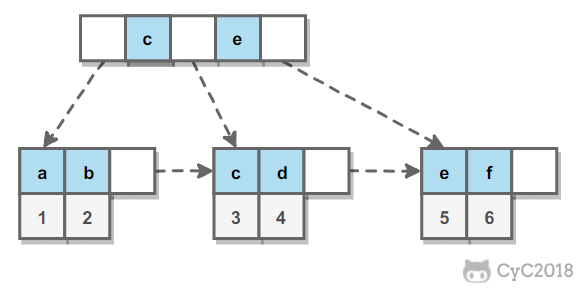
可以指定多个列作为索引列，多个索引列共同组成键。

适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。如果不是按照索引列的顺序进行查找，则无法使用索引。

InnoDB 的 B+Tree 索引分为主索引和辅助索引。主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为聚簇索引。因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。



辅助索引的叶子节点的 data 域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行查找。



#### 哈希索引

哈希索引能够以O(1)时间进行查找，但是失去了有序性：

* 无法用于排序与分组
* 只支持精确查找，无法用于部分查找和范围查找

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找。

#### 全文索引

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较是否相等。

查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

全文索引使用倒排索引实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

#### 空间数据索引

MyISAM存储引擎支持空间数据索引（R-Tree），可以用于地理数据存储，空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用GIS相关的函数来维护数据。

### 索引优化

#### 独立的列

在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引。

例如下面的查询不能使用actor\_id列的索引：

|  |
| --- |
| SELECT actor\_id FROM sakila.actor WHERE actor\_id + 1 = 5; |

#### 多列索引

在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。例如下面语句中，最好把actor\_id和film\_id设置为多列索引。

|  |
| --- |
| SELECT film\_id, actor\_ id FROM sakila.film\_actor  WHERE actor\_id = 1 AND film\_id = 1; |

#### 索引列的顺序

让选择性最强的索引列放在前面。

索引的选择性是指：不重复的索引值和记录总数的比值。最大值为1，此时每个记录都有唯一的索引与其对应。选择性越高，每个记录的区分度越高，查询效率也越高。

例如下面显示的结果中customer\_id的选择性比staff\_id更高，因此最好把customer\_id列放在多列索引的前面。

|  |
| --- |
| SELECT COUNT(DISTINCT staff\_id)/COUNT(\*) AS staff\_id\_selectivity,  COUNT(DISTINCT customer\_id)/COUNT(\*) AS customer\_id\_selectivity,  COUNT(\*)  FROM payment;  结果：  staff\_id\_selectivity: 0.0001  customer\_id\_selectivity: 0.0373  COUNT(\*): 16049 |

#### 前缀索引

对于BLOB、TEXT和VARCHAR类型的列，必须使用前缀索引，只索引开始的部分字符。

前缀长度的选取需要根据索引选择性来确定。

#### 覆盖索引

索引包含所有需要查询的字段的值：

具有以下优点：

* 索引通常远远小于数据行的大小，只读取索引能大大减少数据访问量。
* 一些存储引擎（例如MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
* 对于InnoDB引擎，若辅助索引能够覆盖查询，则无需访问主索引。

### 索引的优点

* 大大减少了服务器需要扫描的数据行数；
* 帮助服务器避免进行排序和分组，以及避免创建临时表（B+Tree索引是有序的，可以用于ORDER BY和GROUP BY操作。临时表主要是在排序和分组过程中创建，不需要排序和分组，也就不需要创建临时表）。
* 将随机I/O变为顺序I/O（B+Tree索引是有序的，会将相邻的数据都存储在一起）。

### 索引的使用条件

* 对于非常小的表，大部分情况下简单的全表扫描比建立索引更高效。
* 对于中到大型的表，索引就非常有效。
* 但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术。

## 查找性能优化

### 使用Explain进行分析

Explain用来分析SELECT查询语句，开发人员可以通过分析Explain结果来优化查询语句。

比较重要的字段有：

* select\_type：查询类型，有简单查询、联合查询、子查询等；
* key：使用的索引；
* rows：扫描的行数。

### 优化数据访问

#### 减少请求的数据量

* 只返回必要的列：最好不要使用SELECT \* 语句。
* 只返回必要的行：使用LIMIT语句来限制返回的数据。
* 缓存重复查询的数据：使用缓存可以避免在数据库中进行查询，特别在要查询的数据经常被重复查询时，缓存带来的查询性能提升将会是非常明显的。

#### 减少服务器端扫面的行数

最有效的方式是使用索引来覆盖查询。

### 重构查询方式

#### 切分大查询

一个大查询如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

|  |
| --- |
| DELETE FROM messages WHERE create < DATE\_SUB(NOW(), INTERVAL 3 MONTH); |

|  |
| --- |
| rows\_affected = 0  do {  rows\_affected = do\_query(  "DELETE FROM messages WHERE create < DATE\_SUB(NOW(), INTERVAL 3 MONTH) LIMIT 10000")  } while rows\_affected > 0 |

#### 分解大连接查询

将一个大连接查询分解成对每一个表进行一次单表查询，然后在应用程序中进行关联，这样做的好处有：

* 让缓存更高效。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。儿分解后的多个查询，即时其中一个表发生变化，对其它表的查询缓存依然可以使用；
* 分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询。
* 减少锁竞争；
* 在应用层进行连接，可以更容易对数据库进行拆分，从而更容易做到高性能和可伸缩。
* 查询本身效率也可能会有所提升。例如下面的例子中，使用IN()代替连接查询，可以让MySQL按照ID顺序进行查询，这可能比随机的连接要更高效。

|  |
| --- |
| SELECT \* FROM tab  JOIN tag\_post ON tag\_post.tag\_id=tag.id  JOIN post ON tag\_post.post\_id=post.id  WHERE tag.tag='mysql'; |

|  |
| --- |
| SELECT \* FROM tag WHERE tag='mysql';  SELECT \* FROM tag\_post WHERE tag\_id=1234;  SELECT \* FROM post WHERE post.id IN (123,456,567,9098,8904); |

## 存储引擎

### InnoDB

InnoDB是MySQL默认的事务型存储引擎，只有在需要它不支持的特性时，才考虑使用其它存储引擎。

实现了四个标准的隔离级别，默认级别是可重复读（REPEATABLE READ）。在可重复读隔离级别下，通过多版本并发控制（MVCC）+间隙锁（Next-Key Locking）防止幻影读。

主索引是聚簇索引，在索引中保存了数据，从而避免直接读取磁盘，因此对查询性能有很大的提升。

内部做了很多优化，包括从磁盘读取数据时采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。

支持真正的在线热备份。其他存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

### MyISAM

设计简单，数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用它。

提供了大量的特性，包括压缩表、空间数据索引等。

不支持事务。

不支持行级锁，只能对整张表加锁，读取时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但在表有读取操作的同时，也可以往表中插入新的记录，这被称为并发插入（CONCURRENT INSERT）。

可以手工或者自动执行检查和修复操作，但是和事务恢复以及崩溃恢复不同，可能导致一些数据丢失，而且修复操作是非常慢的。

如果指定了DELAY\_KEY\_WRITE选项，在每次修改执行完成时，不会立即将修改的索引数据写入磁盘，而是会写到内存中的键缓冲区，只有在清理键缓冲区或者关闭表的时候才会将对应的索引块写入磁盘。这种方式可以极大的提升写入性能，但是在数据库或者主机崩溃时会造成索引损坏，需要执行修复操作。

### 比较

* 事务：InnoDB是事务型的，可以使用Commit和Rollback语句。
* 并发：MyISAM只支持表级锁，而InnoDB还支持行级锁。
* 外键：InnoDB支持外键。
* 备份：InnoDB支持在线热备份。
* 崩溃恢复：MyISAM崩溃后发生损坏的概率比InnoDB高很多，而且恢复的速度也更慢。
* 其它特性：MyISAM支持压缩表和空间索引。

## 数据类型

### 整型

TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT分别使用8，16，24，32，64位存储空间，一般情况下越小的列越好。

INT（11）中的数字只是规定了交互工具显示字符的个数，对于存储和计算来说是没有意义的。

### 浮点数

FLOAT和DOUBLE为浮点类型，DECIMAL为高精度小数类型。CPU原生支持浮点运算，但是不支持DECIMAL类型的计算，因此DECIMAL的计算比浮点类型需要更高的代价。

FLOAT、DOUBLE和DECIMAL都可以指定列宽，例如DECIMAL（18, 9）表示总共18位，取9位存储小数部分，剩下9位存储整数部分。

### 字符串

主要有CHAR和VARCHAR两种类型：一种是定长的，一种是变长的。

VARCHAR这种变长类型能够节省空间，因为只需要存储必要的内容。但在执行UPDATE时可能会使行变得比原来长，当超出一个页所能容纳的大小时，就要执行额外的操作。MyISAM会将行拆成不同的片段存储，而InnoDB则需要分裂页来使行放进页内。

在进行存储和检索时，会保留VARCHAR末尾的空格，而会删除CHAR末尾的空格。

### 时间和日期

MySQL提供了两种相似的日期时间类型：DATETIME和TIMESTAMP。

#### DATETIME

能够保存从1000年到9999年的日期和时间，精度为秒，使用8字节的存储空间。

它与时区无关。

默认情况下，MySQL以一种可排序的、无歧义的格式显示DATETIME值，例如：“2008-01-16 22:37:08”，这是ANSI标准定义的日期和时间表示方法。

#### TIMESTAMP

和UNIX时间戳相同，保存从1970年1月1日午夜（格林威治时间）以来的秒数，使用4字节，只能表示从1970年到2038年。

它和时区有关，也就是说一个时间戳在不同的时区所代表的具体时间是不同的。

MySQL提供了FORM\_UNIXTIME()函数把UNIX时间戳转换为日期，并且提供了UNIX\_TIMESTAMP()函数把日期转换为UNIX时间戳。

默认情况下，如果插入时没有指定TIMESTAMP列的值，会将这个值设置为当前时间。

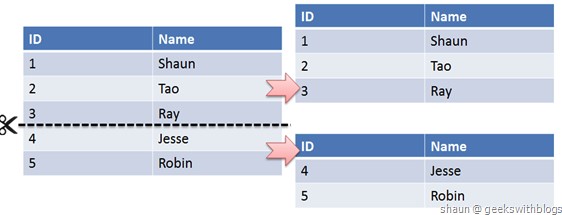
应该尽量使用TIMESTAMP，因为它比DATETIME空间效率更高。

## 切分

### 水平切分

水平切分又称为Sharing，它是将同一个表中的记录拆分到多个结构相同的表中。

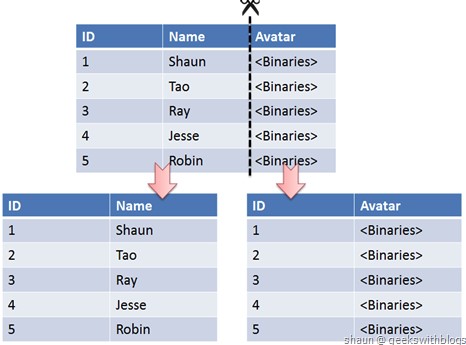
当一个表的数据不断增多时，Sharing是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。



### 垂直切分

垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。

在数据库的层面使用垂直切分将按照数据库中表的密集程度部署到不同的库中，例如将原来的电商数据库垂直切分成商品数据库、用户数据库等。



### Sharing策略

* 哈希取模：hash(key) % N;
* 范围：可以是ID范围也可以是时间范围；
* 映射表：使用单独的一个数据库来存储映射关系。

### Sharing存在的问题

#### 事务问题

使用分布式事务来解决，如：XA接口。

#### 连接

可以将原来的连接分解成多个单表查询，然后在用户程序中进行连接。

#### ID唯一性

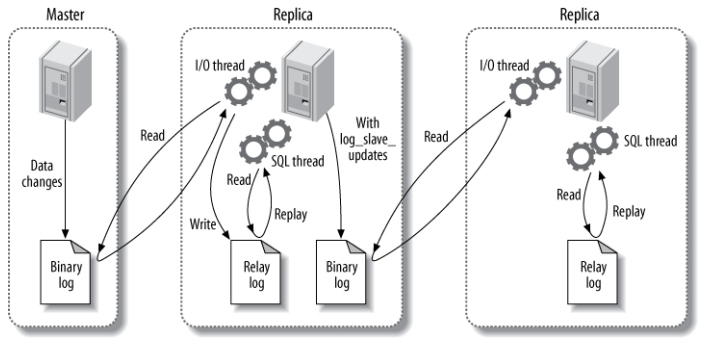
* 使用全局唯一ID（GUID）；
* 为每个分片指定一个ID范围；
* 分布式ID生成器（如Twitter的Snowflake算法）

## 复制

### 主从复制

主要涉及三个线程：binlog线程、I/O线程和SQL线程。

* binlog线程：负责将主服务器上的数据更改写入二进制日志（Binary Log）中。
* I/O线程：负责从主服务器上读取二进制日志，并写入从服务器的中继日志（Relay Log）。
* SQL线程：负责读取中继日志，解析出主服务器已经执行的数据更改并在从服务器中重放（Replay）。



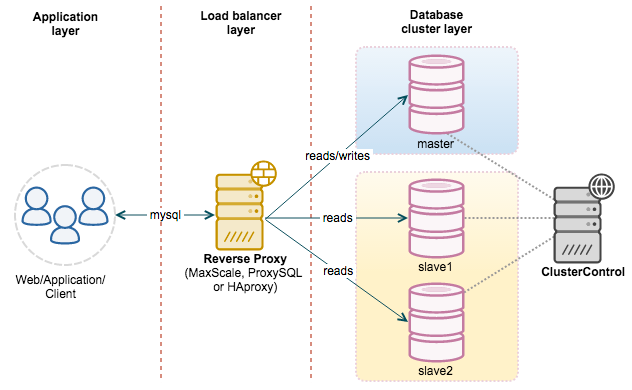
### 读写分离

主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

读写分离能提高性能的原因在于：

* 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
* 从服务器可以使用MyISAM，提升查询性能以及节约系统开销；
* 增加冗余，提高可用性。

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。

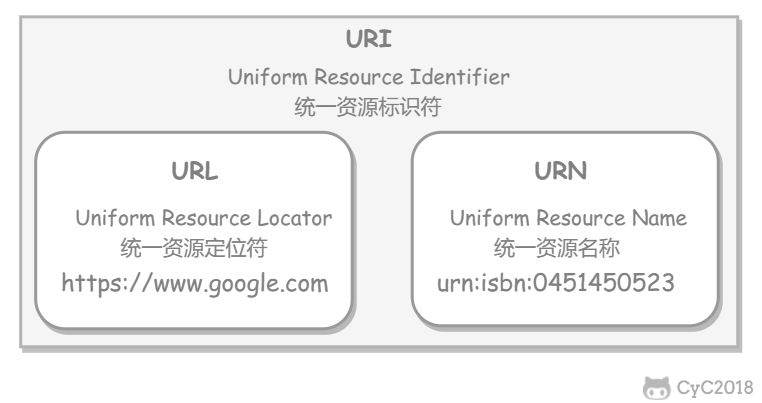


# HTTP

## 基础概念

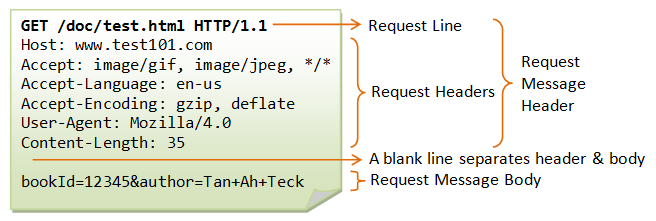
### URI

URI 包含 URL 和 URN。

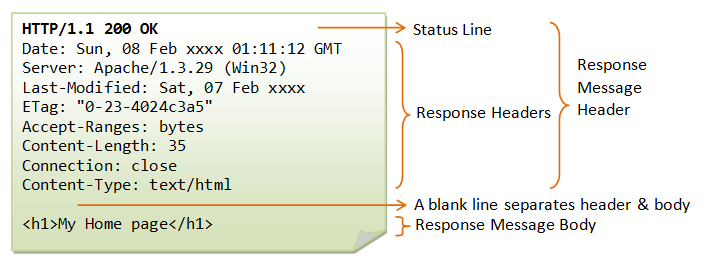


### 请求和响应报文

#### 请求报文



#### 响应报文



## HTTP方法

客户端发送的 请求报文 第一行为请求行，包含了方法字段。

### GET

作用：获取资源

当前网络请求中，绝大部分使用的是GET方法

### HEAD

作用：获取报文首部

主要用于确认URL的有效性以及资源更新的日期时间等

### POST

作用：传输实体主体

更多POST与GET的比较请见第九章

### PUT

作用：上传文件

由于自身不带验证机制，任何人都可以上传文件，因此存在安全性问题，一般不使用该方法。

|  |
| --- |
| PUT /new.html HTTP/1.1  Host: example.com  Content-type: text/html  Content-length: 16  <p>New File</p> |

### PATCH

作用：对资源进行部分修改

PUT也可以用于修改资源，但是只能完全替代原始资源，PATCH允许部分修改。

|  |
| --- |
| PATCH /file.txt HTTP/1.1  Host: www.example.com  Content-Type: application/example  If-Match: "e0023aa4e"  Content-Length: 100  [description of changes] |

### DELETE

作用：删除文件

与PUT功能相反，并且同样不带验证机制。

|  |
| --- |
| DELETE /file.html HTTP/1.1 |

### OPTIONS

作用：查询支持的方法

查询指定的URL能够支持的方法

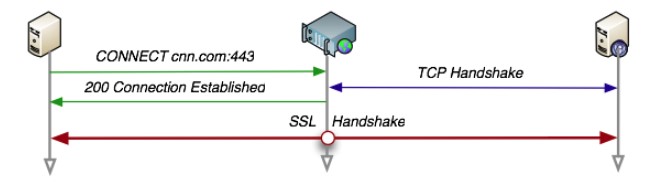
会返回Allow：GET、POST、HEAD、OPTIONS这样的内容。

### CONNECT

作用：要求在与代理服务器通信时建立隧道

使用 SSL（Secure Sockets Layer，安全套接层）和 TLS（Transport Layer Security，传输层安全）协议把通信内容加密后经网络隧道传输。

|  |
| --- |
| CONNECT www.example.com:443 HTTP/1.1 |



### TRACE

作用：追踪路径

服务器会将通信路径返回给客户端

发送请求时，在Max-Forwards首部字段中填入数值，每经过一个服务器就会减少1，当数值为0时就停止传输。

通常不会使用TRACE，并且它容易受到XST攻击（Cross-Site Tracing，跨站追踪）。

## HTTP状态码

服务器返回的响应报文中第一行为状态行，包含了状态码以及原因短语，用来告知客户端请求的结果。

|  |  |  |
| --- | --- | --- |
| **状态码** | **类别** | **含义** |
| 1XX | Informational（信息性状态码） | 接收的请求正在处理 |
| 2XX | Success（成功状态码） | 请求正常处理完毕 |
| 3XX | Redirection（重定向状态码） | 需要进行附加操作以完成请求 |
| 4XX | Client Error（客户端错误状态码） | 服务器无法处理请求 |
| 5XX | Server Error（服务器错误状态码） | 服务器处理请求出错 |

### 1XX信息

* 100 Continue：表明到目前为止都很正常，客户端可以连续发送请求或者忽略这个响应。

### 2XX成功

* 200 OK
* 204 No Content：请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用。
* 206 Partial Content：表示客户端进行了范围请求，响应报文包含由Content-Range指定范围的实体内容。

### 3XX重定向

* 301 Moved Permanently：永久性重定向
* 302 Found：临时性重定向
* 303 See Other：和302有着相同的功能，但是303明确要求客户端应该采用GET方法获取资源

注：虽然 HTTP 协议规定 301、302 状态下重定向时不允许把 POST 方法改成 GET 方法，但是大多数浏览器都会在 301、302 和 303 状态下的重定向把 POST 方法改成 GET 方法。

* 304 Not Modified：如果请求报文首部包含一些条件，例如：If-Match，If-Modified-Since，If-None-Match，If-Range，If-Unmodified-Since，如果不满足条件，则服务器会返回 304 状态码。
* 307 Temporary Redirect：临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的 POST 方法改成 GET 方法。

### 4XX客户端错误

* 400 Bad Request：请求报文中存才语法错误
* 401 Unauthorized：该状态码表示发送的请求需要有认证信息（BASIC 认证、DIGEST 认证）。如果之前已进行过一次请求，则表示用户认证失败。
* 403 Forbidden：请求被拒绝。
* 404 Not Found

### 5XX服务器错误

* 500 Internal Server Error：服务器正在执行请求时发生错误
* 503 Service Unavailable：服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。

## HTTP首部

有4种类型的首部字段：通用首部字段、请求首部字段、响应首部字段和实体首部字段。

各种首部字段及其含义如下（不需要全记，仅供查阅）：

### 通用首部字段

|  |  |
| --- | --- |
| **首部字段名称** | **说明** |
| Cache-Control | 控制缓存的行为 |
| Connection | 控制不再转发给代理的首部字段，管理持久连接 |
| Date | 创建报文的日期时间 |
| Pragma | 报文指令 |
| Trailer | 报文末端的首部一览 |
| Transfer-Encoding | 指定报文主体的传输编码方式 |
| Upgrade | 升级为其他协议 |
| Via | 代理服务器的相关信息 |
| Warning | 错误通知 |

### 请求首部字段

|  |  |
| --- | --- |
| **首部字段名** | **说明** |
| Accept | 用户代理可处理的媒体类型 |
| Accept-Charset | 优先的字符集 |
| Accept-Encoding | 优先的内容编码 |
| Accept-Language | 优先的语言（自然语言） |
| Authorization | Web 认证信息 |
| Expect | 期待服务器的特定行为 |
| From | 用户的电子邮箱地址 |
| Host | 请求资源所在服务器 |
| If-Match | 比较实体标记（ETag） |
| If-Modified-Since | 比较资源的更新时间 |
| If-None-Match | 比较实体标记（与 If-Match 相反） |
| If-Range | 资源未更新时发送实体 Byte 的范围请求 |
| If-Unmodified-Since | 比较资源的更新时间（与 If-Modified-Since 相反） |
| Max-Forwards | 最大传输逐跳数 |
| Proxy-Authorization | 代理服务器要求客户端的认证信息 |
| Range | 实体的字节范围请求 |
| Referer | 对请求中 URI 的原始获取方 |
| TE | 传输编码的优先级 |
| User-Agent | HTTP 客户端程序的信息 |

### 响应首部字段

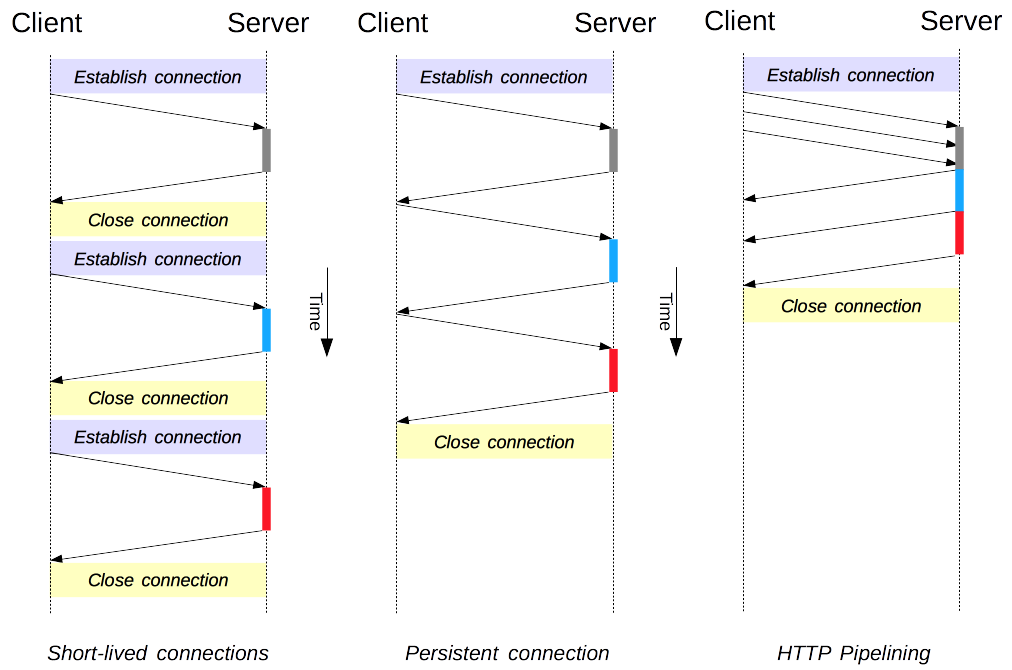
|  |  |
| --- | --- |
| **首部字段名** | **说明** |
| Accept-Ranges | 是否接受字节范围请求 |
| Age | 推算资源创建经过时间 |
| ETag | 资源的匹配信息 |
| Location | 令客户端重定向至指定 URI |
| Proxy-Authenticate | 代理服务器对客户端的认证信息 |
| Retry-After | 对再次发起请求的时机要求 |
| Server | HTTP 服务器的安装信息 |
| Vary | 代理服务器缓存的管理信息 |
| WWW-Authenticate | 服务器对客户端的认证信息 |

### 实体首部字段

|  |  |
| --- | --- |
| **首部字段名** | **说明** |
| Allow | 资源可支持的 HTTP 方法 |
| Content-Encoding | 实体主体适用的编码方式 |
| Content-Language | 实体主体的自然语言 |
| Content-Length | 实体主体的大小 |
| Content-Location | 替代对应资源的 URI |
| Content-MD5 | 实体主体的报文摘要 |
| Content-Range | 实体主体的位置范围 |
| Content-Type | 实体主体的媒体类型 |
| Expires | 实体主体过期的日期时间 |
| Last-Modified | 资源的最后修改日期时间 |

## 具体应用

### 连接管理



#### 短连接与长连接

当浏览器访问一个包含多张图片的 HTML 页面时，除了请求访问的 HTML 页面资源，还会请求图片资源。如果每进行一次 HTTP 通信就要新建一个 TCP 连接，那么开销会很大。

长连接只需要建立一次 TCP 连接就能进行多次 HTTP 通信。

* 从HTTP/1.1开始默认是长连接的，如果要断开连接，需要由客户端或者服务器端提出断开，使用Connection：close；
* 在HTTP/1.1之前默认是短连接的，如果需要使用长连接，则使用Connection：keep-alive。

#### 流水线

默认情况下，HTTP请求是按顺序发出的，下一个请求只有在当前请求收到响应之后才会被发出。由于受到网络延迟和带宽限制，在下一个请求被发送到服务器之前，可能需要等待很长的时间。

流水线是在同一条长连接上连续发出请求，而不用等待响应返回，这样可以减少延迟。

### Cookie

HTTP协议是无状态的，主要为了让HTTP协议尽可能简单，使得它能够处理大量事务。HTTP/1.1引入Cookie来保存状态信息。

Cookie是服务器发送到用户浏览器并保存在本地的一小块数据，它会在浏览器之后向同一服务器再次发起请求时被携带上，用于告知服务端两个请求是否来自同一浏览器。由于之后每次请求都会需要携带Cookie数据，因此会带来额外的性能开销（尤其是在移动环境下）。

Cookie曾一度用于客户端数据的存储，因为当时并没有其它合适的存储方法而作为唯一的存储手段，但现在随着现代浏览器开始支持各种各样的存储方式，Cookie逐渐被淘汰。新的浏览器API已经允许开发者直接将数据存储到本地，如使用Web storage API（本地存储和会话存储）或IndexedDB。

#### 用途

* 会话状态管理（如用户登录状态、购物车、游戏分数或其它需要记录的信息）
* 个性化设置（如用户自定义设置、主题等）
* 浏览器行为跟踪（如跟踪分析用户行为等）

#### 创建过程

服务器发送的响应报文包含Set-Cookie首部字段，客户端得到响应报文后把Cookie内容保存到浏览器中。

|  |
| --- |
| HTTP/1.0 200 OK  Content-type: text/html  Set-Cookie: yummy\_cookie=choco  Set-Cookie: tasty\_cookie=strawberry  [page content] |

客户端之后对同一个服务器发送请求时，会从浏览器中取出Cookie信息并通过Cookie请求首部字段发送给服务器。

|  |
| --- |
| GET /sample\_page.html HTTP/1.1  Host: www.example.org  Cookie: yummy\_cookie=choco; tasty\_cookie=strawberry |

#### 分类

* 会话期Cookie：浏览器关闭之后它会被自动删除，也就是说它仅在会话期内有效。
* 持久性Cookie：指定过期时间（Expires）或者有效期（max-age）之后就成为了持久性的Cookie。

|  |
| --- |
| Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; |

#### 作用域

Domain 标识指定了哪些主机可以接受 Cookie。如果不指定，默认为当前文档的主机（不包含子域名）。如果指定了 Domain，则一般包含子域名。例如，如果设置 Domain=mozilla.org，则 Cookie 也包含在子域名中（如 developer.mozilla.org）。

Path 标识指定了主机下的哪些路径可以接受 Cookie（该 URL 路径必须存在于请求 URL 中）。以字符 %x2F ("/") 作为路径分隔符，子路径也会被匹配。例如，设置 Path=/docs，则以下地址都会匹配：

* /docs
* /docs/Web/
* /docs/Web/HTTP

#### JavaScript

浏览器通过 document.cookie 属性可创建新的 Cookie，也可通过该属性访问非 HttpOnly 标记的 Cookie。

|  |
| --- |
| document.cookie = "yummy\_cookie=choco";  document.cookie = "tasty\_cookie=strawberry";  console.log(document.cookie); |

#### HttpOnly

标记为 HttpOnly 的 Cookie 不能被 JavaScript 脚本调用。跨站脚本攻击 (XSS) 常常使用 JavaScript 的 document.cookie API 窃取用户的 Cookie 信息，因此使用 HttpOnly 标记可以在一定程度上避免 XSS 攻击。

|  |
| --- |
| Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly |

#### Secure

标记为 Secure 的 Cookie 只能通过被 HTTPS 协议加密过的请求发送给服务端。但即便设置了 Secure 标记，敏感信息也不应该通过 Cookie 传输，因为 Cookie 有其固有的不安全性，Secure 标记也无法提供确实的安全保障。

#### Session

除了可以将用户信息通过 Cookie 存储在用户浏览器中，也可以利用 Session 存储在服务器端，存储在服务器端的信息更加安全。

Session 可以存储在服务器上的文件、数据库或者内存中。也可以将 Session 存储在 Redis 这种内存型数据库中，效率会更高。

使用 Session 维护用户登录状态的过程如下：

* 用户进行登录时，用户提交包含用户名和密码的表单，放入 HTTP 请求报文中；
* 服务器验证该用户名和密码，如果正确则把用户信息存储到 Redis 中，它在 Redis 中的 Key 称为 Session ID；
* 服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie 值存入浏览器中；
* 客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID，从 Redis 中取出用户信息，继续之前的业务操作。

应该注意 Session ID 的安全性问题，不能让它被恶意攻击者轻易获取，那么就不能产生一个容易被猜到的 Session ID 值。此外，还需要经常重新生成 Session ID。在对安全性要求极高的场景下，例如转账等操作，除了使用 Session 管理用户状态之外，还需要对用户进行重新验证，比如重新输入密码，或者使用短信验证码等方式。

#### 浏览器禁用Cookie

此时无法使用 Cookie 来保存用户信息，只能使用 Session。除此之外，不能再将 Session ID 存放到 Cookie 中，而是使用 URL 重写技术，将 Session ID 作为 URL 的参数进行传递。

#### Cookie与Session选择

* Cookie 只能存储 ASCII 码字符串，而 Session 则可以存储任何类型的数据，因此在考虑数据复杂性时首选 Session；
* Cookie 存储在浏览器中，容易被恶意查看。如果非要将一些隐私数据存在 Cookie 中，可以将 Cookie 值进行加密，然后在服务器进行解密；
* 对于大型网站，如果用户所有的信息都存储在 Session 中，那么开销是非常大的，因此不建议将所有的用户信息都存储到 Session 中。

### 缓存

#### 优点

* 缓解服务器压力；
* 降低客户端获取资源的延迟，缓存通常位于内存中，读取缓存的速度更快。并且缓存服务器在地理位置上也有可能比源服务器来得近，例如浏览器缓存。

#### 实现方法

* 让代理服务器进行缓存；
* 让客户端浏览器进行缓存

#### Cache-Control

HTTP/1.1 通过Cache-Control首部字段来控制缓存。

##### 禁止进行缓存

no-store指令规定不能对请求或响应的任何一部分进行缓存。

##### 强制确认缓存

no-cache 指令规定缓存服务器需要先向源服务器验证缓存资源的有效性，只有当缓存资源有效时才能使用该缓存对客户端的请求进行响应。

##### 私有缓存和公共缓存

private指令规定了将资源作为私有缓存，只能被单独用户使用，一般存储在用户浏览器中。

public 指令规定了将资源作为公共缓存，可以被多个用户使用，一般存储在代理服务器中。

##### 缓存过期机制

max-age 指令出现在请求报文，并且缓存资源的缓存时间小于该指令指定的时间，那么就能接受该缓存。

max-age 指令出现在响应报文，表示缓存资源在缓存服务器中保存的时间。

Expires 首部字段也可以用于告知缓存服务器该资源什么时候会过期。

* 在 HTTP/1.1 中，会优先处理 max-age 指令；
* 在 HTTP/1.0 中，max-age 指令会被忽略掉。

#### 缓存验证

需要先了解ETag首部字段的含义，它是资源的唯一标识。URL不能唯一表示资源，例如http://www.google.com/ 有中文和英文两个资源，只有 ETag 才能对这两个资源进行唯一标识。

|  |
| --- |
| ETag: "82e22293907ce725faf67773957acd12" |

可以将缓存资源的 ETag 值放入 If-None-Match 首部，服务器收到该请求后，判断缓存资源的 ETag 值和资源的最新 ETag 值是否一致，如果一致则表示缓存资源有效，返回 304 Not Modified。

|  |
| --- |
| If-None-Match: "82e22293907ce725faf67773957acd12" |

Last-Modified首部字段也可以用于缓存验证，它包含在源服务器发送的响应报文中，指示源服务器对资源的最后修改时间。但是它是一种弱校验器，因为只能精确到一秒，所以它通常作为 ETag 的备用方案。如果响应首部字段里含有这个信息，客户端可以在后续的请求中带上 If-Modified-Since 来验证缓存。服务器只在所请求的资源在给定的日期时间之后对内容进行过修改的情况下才会将资源返回，状态码为 200 OK。如果请求的资源从那时起未经修改，那么返回一个不带有实体主体的 304 Not Modified 响应报文。

|  |
| --- |
| Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT |

|  |
| --- |
| If-Modified-Since: Wed, 21 Oct 2015 07:28:00 GMT |

### 内容协商

通过内容协商返回最合适的内容，例如根据浏览器的默认语言选择返回中文界面还是英文界面。

#### 类型

##### 服务端驱动型

客户端设置特定的HTTP首部字段，例如Accept、Accept-Charset、Accept-Encoding、Accept-Language，服务器根据这些字段返回特定的资源。

它存在以下问题：

* 服务器很难知道客户端浏览器的全部信息；
* 客户端提供的信息相当冗长（HTTP/2协议的首部压缩机制缓解了这个问题），并且存在隐私风险（HTTP指纹识别技术）。
* 给定的资源需要返回不同的展现形式，共享缓存的效率会降低，而服务器端的实现会越来越复杂。

##### 代理驱动型

服务器返回300 Multiple Choices或者406 Not Acceptable，客户端从中选出最合适的那个资源。

#### Vary

在使用内容协商的情况下，只有当缓存服务器中的缓存满足内容协商条件时，才能使用该缓存，否则应该向源服务器请求该资源。

例如，一个客户端发送了一个包含 Accept-Language 首部字段的请求之后，源服务器返回的响应包含 Vary: Accept-Language 内容，缓存服务器对这个响应进行缓存之后，在客户端下一次访问同一个 URL 资源，并且 Accept-Language 与缓存中的对应的值相同时才会返回该缓存。

### 内容编码

内容编码将实体主体进行压缩，从而减少传输的数据量。

常用的内容编码有：gzip、compress、deflate、identity。

浏览器发送 Accept-Encoding 首部，其中包含有它所支持的压缩算法，以及各自的优先级。服务器则从中选择一种，使用该算法对响应的消息主体进行压缩，并且发送 Content-Encoding 首部来告知浏览器它选择了哪一种算法。由于该内容协商过程是基于编码类型来选择资源的展现形式的，响应报文的 Vary 首部字段至少要包含 Content-Encoding。

### 范围请求

如果网络出现中断，服务器只发送了一部分数据，范围请求可以使得客户端只请求服务器未发送的那部分数据，从而避免服务器重新发送所有数据。

#### Range

在请求报文中添加 Range 首部字段指定请求的范围。

|  |
| --- |
| GET /z4d4kWk.jpg HTTP/1.1  Host: i.imgur.com  Range: bytes=0-1023 |

请求成功的话服务器返回的响应包含 206 Partial Content 状态码。

|  |
| --- |
| HTTP/1.1 206 Partial Content  Content-Range: bytes 0-1023/146515  Content-Length: 1024  ...  (binary content) |

#### Accept-Ranges

响应首部字段 Accept-Ranges 用于告知客户端是否能处理范围请求，可以处理使用 bytes，否则使用 none。

|  |
| --- |
| Accept-Ranges: bytes |

#### 响应状态码

* 在请求成功的情况下，服务器会返回206 Partial Content状态码。
* 在请求的范围越界的情况下，服务器会返回416 Requested Range Not Satisfiable状态码。
* 在不支持范围请求的情况下，服务器会返回200 OK状态码。

### 分块传输编码

Chunked Transfer Encoding，可以把数据分割成多块，让浏览器逐步显示页面。

### 多部分对象集合

一份报文主体内可含有多种类型的实体同时发送，每个部分之间用 boundary 字段定义的分隔符进行分隔，每个部分都可以有首部字段。

例如，上传多个表单时可以使用如下方式：

|  |
| --- |
| Content-Type: multipart/form-data; boundary=AaB03x  --AaB03x  Content-Disposition: form-data; name="submit-name"  Larry  --AaB03x  Content-Disposition: form-data; name="files"; filename="file1.txt"  Content-Type: text/plain  ... contents of file1.txt ...  --AaB03x-- |

### 虚拟主机

HTTP/1.1 使用虚拟主机技术，使得一台服务器拥有多个域名，并且在逻辑上可以看成多个服务器。

### 通信数据转发

#### 代理

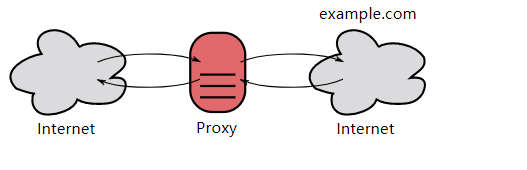
代理服务器接受客户端的请求，并且转发给其它服务器。

使用代理的主要目的是：

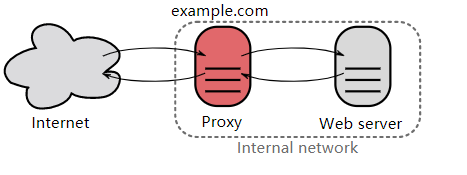
* 缓存；
* 负载均衡；
* 网络访问控制；
* 访问日志记录。

代理服务器分为正向代理和反向代理两种：

* 用户察觉得到正向代理的存在。



* 而反向代理一般位于内部网络中，用户觉察不到。



#### 网关

与代理服务器不同的是，网关服务器会将 HTTP 转化为其它协议进行通信，从而请求其它非 HTTP 服务器的服务。

#### 隧道

使用 SSL 等加密手段，在客户端和服务器之间建立一条安全的通信线路。

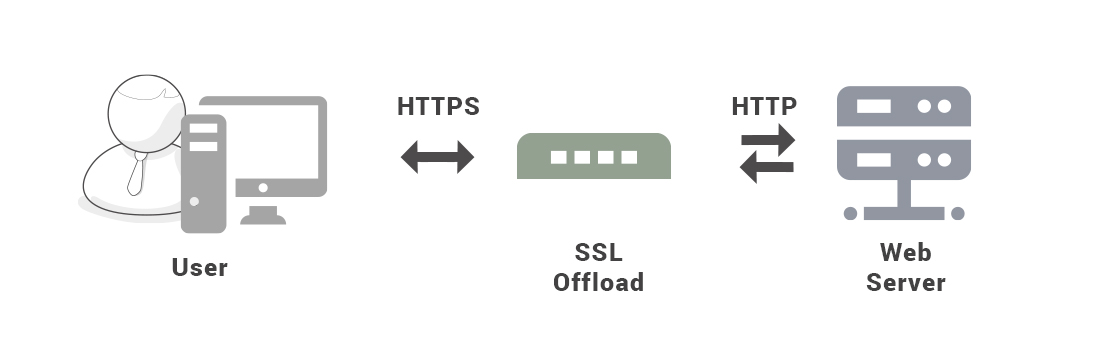
## HTTPS

HTTP有以下安全性问题：

* 使用明文进行通信，内容可能会被窃听；
* 不验证通信方的身份，通信方的身份有可能遭遇伪装；
* 无法证明报文的完整性，报文可能被篡改。

HTTPS并不是新协议，而是让HTTP先和SSL（Secure Sockets Layer）通信，再由SSL和TCP通信，也就是说HTTPS使用了隧道进行通信。

通过使用SSL，HTTPS具有了加密（防窃听），认证（防伪装）和完整性保护（防篡改）。

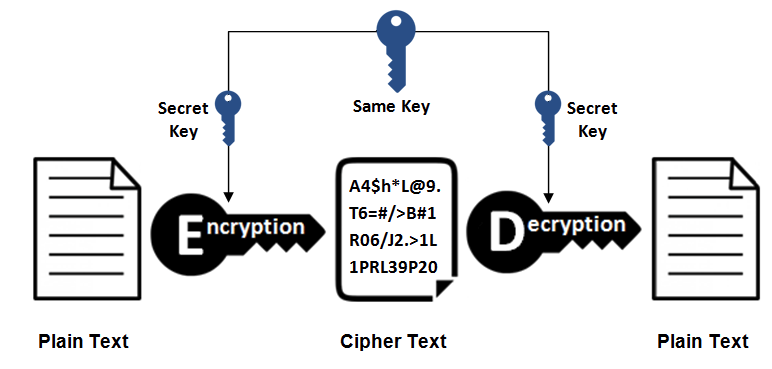


### 加密

#### 对称密钥加密

对称密钥加密（Symmetric-Key Encryption），加密和解密使用同一密钥。

* 优点：运算速度快；
* 缺点：无法安全的将密钥传输给通信方。



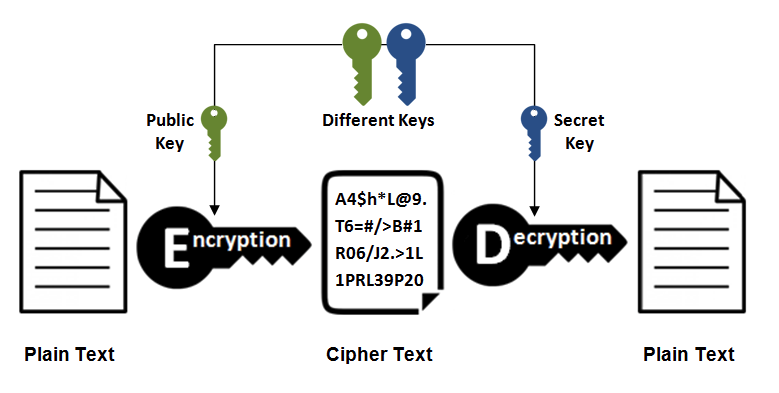
#### 非对称密钥加密

非对称密钥加密，又称公开密钥加密（Public-Key Encryption），加密和解密使用不同的密钥。

公开密钥所有人都可以获得，通信发送方获得接收方的公开密钥之后，就可以使用公开密钥进行加密，接收方收到通信内容后使用私有密钥解密。

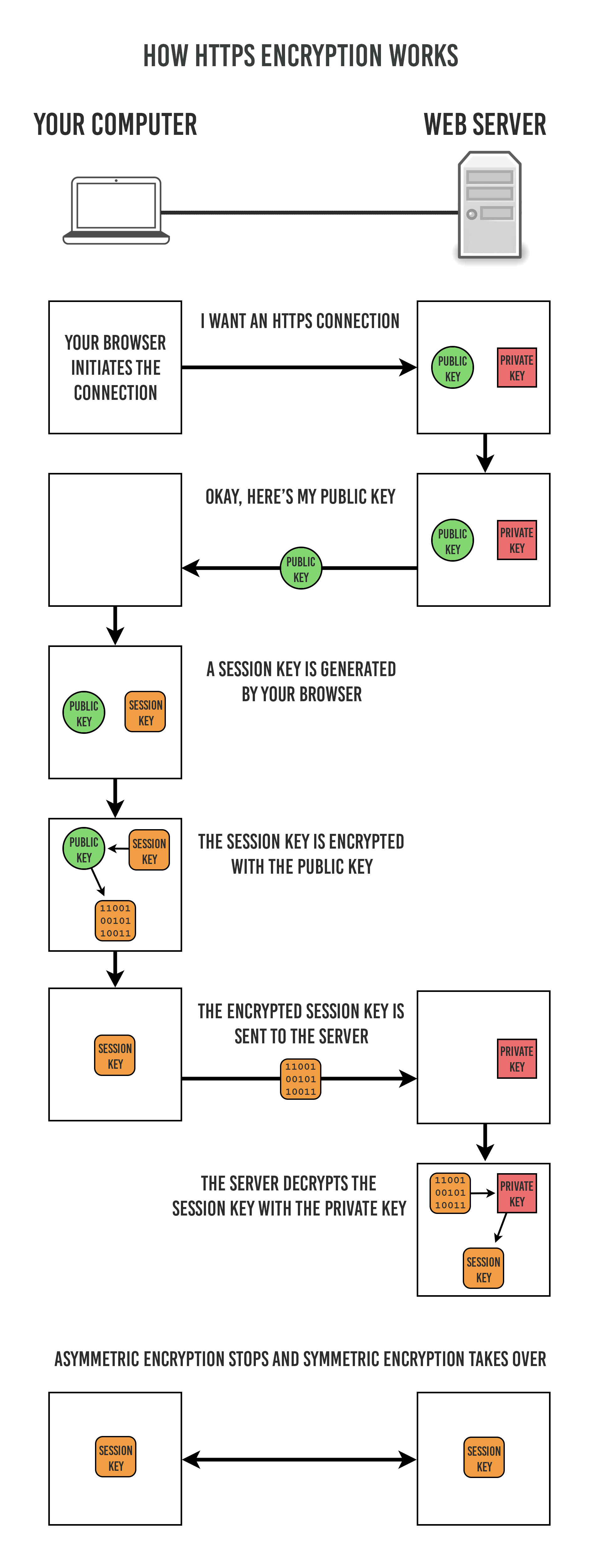
非对称密钥除了用来加密，还可以用来进行签名。因为私有密钥无法被其他人获取，因此通信发送方使用其私有密钥进行签名，通信接收方使用发送方的公开密钥对签名进行解密，就能判断这个签名是否正确。

* 优点：可以更安全的将公开密钥传输给通信发送方；
* 缺点：运算速度慢



#### HTTPS采用的加密方式

HTTPS采用混合的加密机制，使用非对称密钥加密用于传输对称密钥来保证传输过程的安全性，之后使用对称密钥加密进行通信来保证通信过程的效率。（下图中的Session Key就是对称密钥）



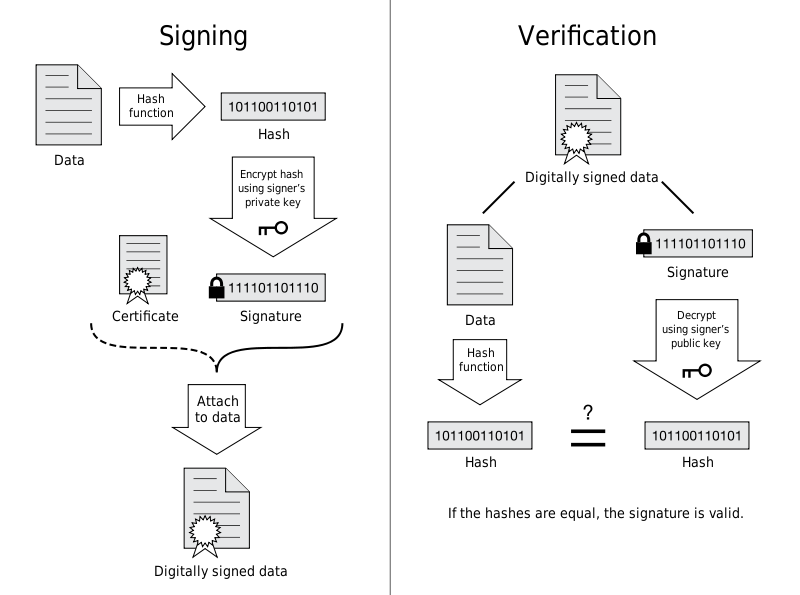
### 认证

通过使用认证来对通信方进行认证。

数字证书认证机构（CA，Certificate Authority）是客户端与服务器双方都可信赖的第三方机构。

服务器的运营人员向 CA 提出公开密钥的申请，CA 在判明提出申请者的身份之后，会对已申请的公开密钥做数字签名，然后分配这个已签名的公开密钥，并将该公开密钥放入公开密钥证书后绑定在一起。

进行 HTTPS 通信时，服务器会把证书发送给客户端。客户端取得其中的公开密钥之后，先使用数字签名进行验证，如果验证通过，就可以开始通信了。



### 完整性保护

SSL提供报文摘要功能来进行完整性保护。

HTTP也提供了MD5报文摘要功能，但不是安全的。例如报文内容被篡改之后，同时重新计算MD5的值，通信接收方是无法意识到发生了篡改。

HTTPS的报文摘要功能之所以安全，是因为它结合了加密和认证这两个操作。试想一下，加密之后的报文，遭到篡改之后，也很难重新计算报文摘要，因为无法轻易获取明文。

### HTTPS的缺点

* 因为需要进行加密解密等过程，因此速度会更慢；
* 需要支付证书授权的高额费用。

## HTTP/2.0

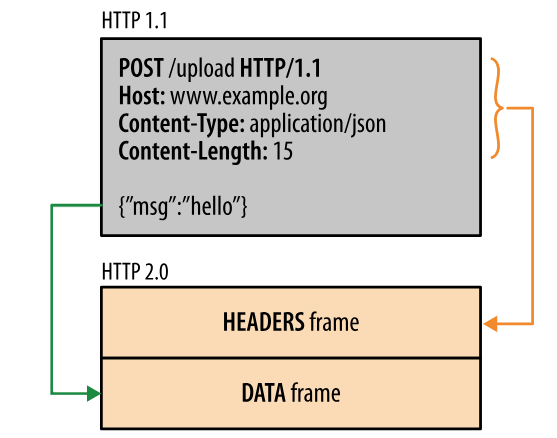
### HTTP/1.x缺陷

HTTP/1.x实现简单是以牺牲性能为代价的：

* 客户端需要使用多个连接才能实现并发和缩短延迟；
* 不会压缩请求和响应首部，从而导致不必要的网络流量；
* 不支持有效的资源优先级，致使底层 TCP 连接的利用率低下。

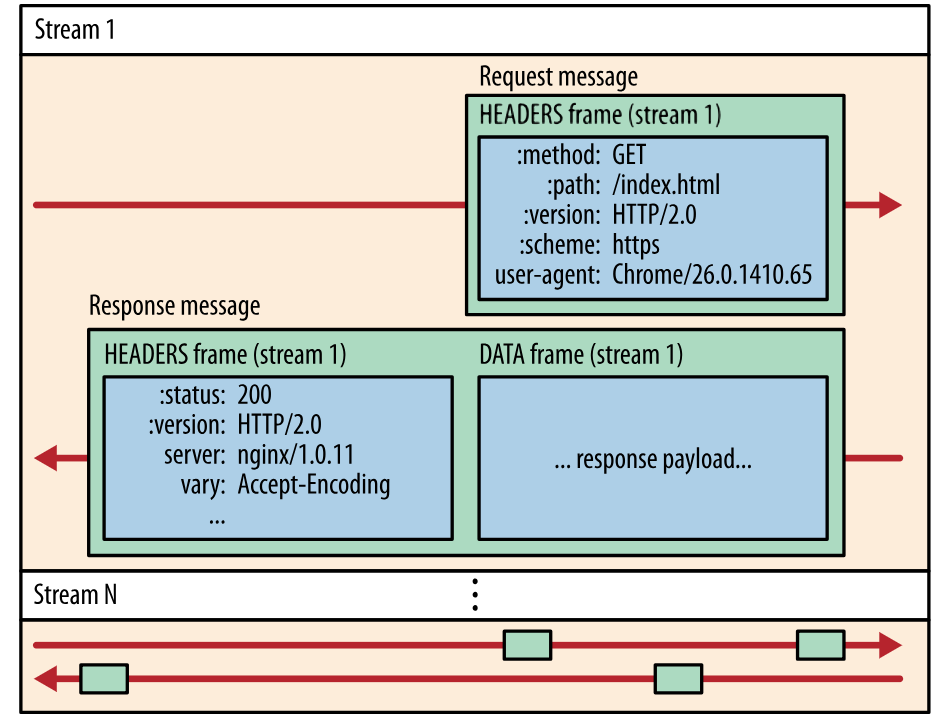
### 二进制分帧层

HTTP/2.0 将报文分成 HEADERS 帧和 DATA 帧，它们都是二进制格式的。



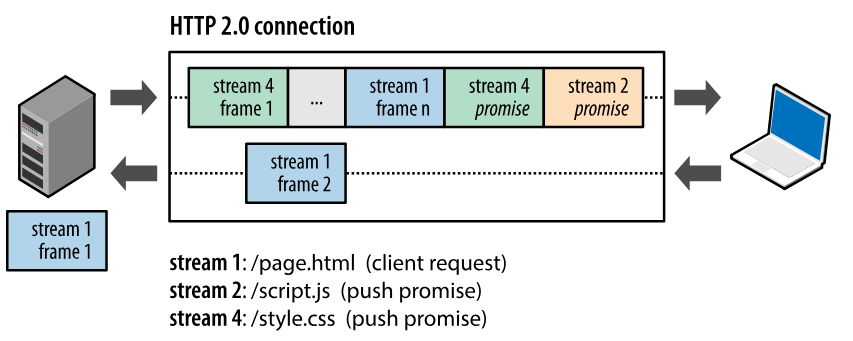
在通信过程中，只会有一个 TCP 连接存在，它承载了任意数量的双向数据流（Stream）。

* 一个数据流（Stream）都有一个唯一标识符和可选的优先级信息，用于承载双向信息。
* 消息（Message）是与逻辑请求或响应对应的完整的一系列帧。
* 帧（Frame）是最小的通信单位，来自不同数据流的帧可以交错发送，然后再根据每个帧头的数据流标识符重新组装。



### 服务端推送

HTTP/2.0在客户端请求一个资源时，会把相关的资源一起发送给客户端，客户端就不需要再次发起请求了。例如客户端请求 page.html 页面，服务端就把 script.js 和 style.css 等与之相关的资源一起发给客户端。

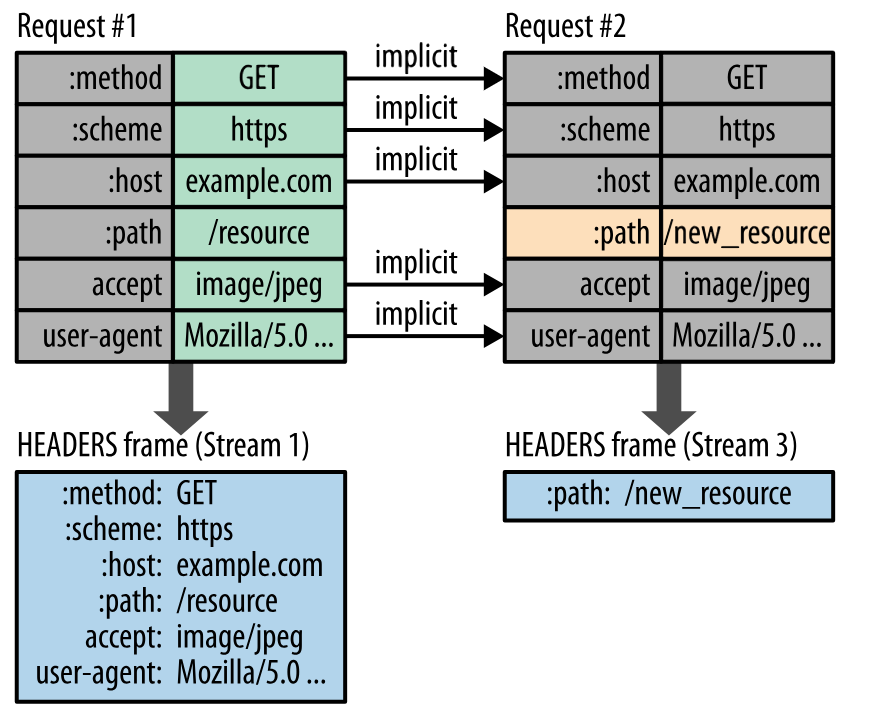


### 首部压缩

HTTP/1.1 的首部带有大量信息，而且每次都要重复发送。

HTTP/2.0 要求客户端和服务器同时维护和更新一个包含之前见过的首部字段表，从而避免了重复传输。

不仅如此，HTTP/2.0 也使用 Huffman 编码对首部字段进行压缩。



## HTTP/1.1新特性

* 默认是长连接
* 支持流水线
* 支持同时打开多个TCP连接
* 支持虚拟主机
* 新增状态码100
* 支持分块传输编码
* 新增缓存处理指令max-age

## GET和POST比较

### 作用

GET用于获取资源，而POST用于传输实体主体

### 参数

GET和POST的请求都能使用额外的参数，但是GET的参数是以查询字符串出现在URL中，而POST的参数存储在实体主体中。不能因为POST参数存储在实体中就认为它的安全性更高，因为照样可以通过一些抓包工具（Fiddler）查看。

因为URL只支持ASCII码，因此GET的参数中如果存在中文等字符就需要先进行编码。例如：中文会转换为%E4%B8%AD%E6%96%87，而空格会转换为%20。POST参数支持标准字符集。

|  |
| --- |
| GET /test/demo\_form.asp?name1=value1&name2=value2 HTTP/1.1 |

|  |
| --- |
| POST /test/demo\_form.asp HTTP/1.1  Host: w3schools.com  name1=value1&name2=value2 |

### 安全

安全的HTTP方法不会改变服务器状态，也就是说它只是可读的。

GET方法是安全的，而POST却不是，因为POST的目的是传送实体主体内容，这个内容可能是用户上传的表单数据，上传成功之后，服务器可能把这个数据存储到数据库中，因此状态也就发生了改变。

安全的方法除了GET之外还有：HEAD、OPTIONS。

不安全的方法除了POST之外还有PUT、DELETE。

### 幂等性

幂等的HTTP方法，同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样的。换句话说就是，幂等方法不应该具有副作用（统计用途除外）。

所有的安全方法也都是幂等的。

在正确实现的条件下，GET，HEAD，PUT 和 DELETE 等方法都是幂等的，而 POST 方法不是。

GET /pageX HTTP/1.1 是幂等的，连续调用多次，客户端接收到的结果都是一样的：

|  |
| --- |
| GET /pageX HTTP/1.1  GET /pageX HTTP/1.1  GET /pageX HTTP/1.1  GET /pageX HTTP/1.1 |

POST /add\_row HTTP/1.1 不是幂等的，如果调用多次，就会增加多行记录：

|  |
| --- |
| POST /add\_row HTTP/1.1 -> Adds a 1nd row  POST /add\_row HTTP/1.1 -> Adds a 2nd row  POST /add\_row HTTP/1.1 -> Adds a 3rd row |

DELETE /idX/delete HTTP/1.1 是幂等的，即使不同的请求接收到的状态码不一样：

|  |
| --- |
| DELETE /idX/delete HTTP/1.1 -> Returns 200 if idX exists  DELETE /idX/delete HTTP/1.1 -> Returns 404 as it just got deleted  DELETE /idX/delete HTTP/1.1 -> Returns 404 |

### 可缓存

如果要对响应进行缓存，需要满足一下条件：

* 请求报文的 HTTP 方法本身是可缓存的，包括 GET 和 HEAD，但是 PUT 和 DELETE 不可缓存，POST 在多数情况下不可缓存的。
* 响应报文的状态码是可缓存的，包括：200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501
* 响应报文的 Cache-Control 首部字段没有指定不进行缓存

### XMLHttpRequest

为了阐述 POST 和 GET 的另一个区别，需要先了解 XMLHttpRequest：

XMLHttpRequest 是一个 API，它为客户端提供了在客户端和服务器之间传输数据的功能。它提供了一个通过 URL 来获取数据的简单方式，并且不会使整个页面刷新。这使得网页只更新一部分页面而不会打扰到用户。XMLHttpRequest 在 AJAX 中被大量使用。

* 在使用 XMLHttpRequest 的 POST 方法时，浏览器会先发送 Header 再发送 Data。但并不是所有浏览器会这么做，例如火狐就不会。
* 而 GET 方法 Header 和 Data 会一起发送。

# SOCKET

## I/O模型

一个输入操作通常包括两个阶段：

* 等待数据准备好
* 从内核向进程复制数据

对于一个套接字上的输入操作，第一步通常涉及等待数据从网络中到达。当所等待数据到达时，它被复制到内核中的某个缓冲区，第二部就是把数据从内核缓冲区复制到应用进程缓冲区。

Unix有五种I/O模型：

* 阻塞式I/O
* 非阻塞式I/O
* I/O复用（select和poll）
* 信号驱动式I/O（SIGIO）
* 异步I/O（AIO）

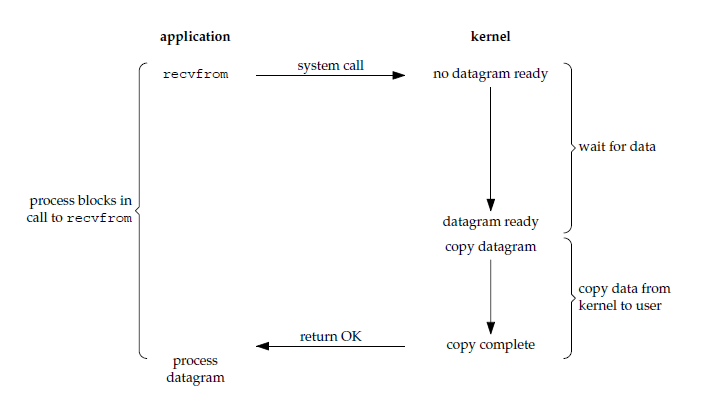
### 阻塞式I/O

应用进程被阻塞，直到数据从内核缓冲区复制到应用进程缓冲区中才返回。

应该注意到，在阻塞的过程中，其它应用进程还可以执行，因此阻塞不意味着整个操作系统都被阻塞。因为其它应用进程还可以执行，所以不消耗 CPU 时间，这种模型的 CPU 利用率会比较高。

下图中，recvfrom() 用于接收 Socket 传来的数据，并复制到应用进程的缓冲区 buf 中。这里把 recvfrom() 当成系统调用。

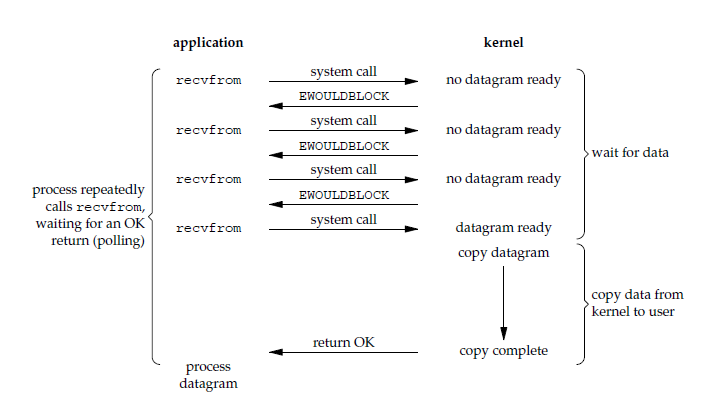
|  |
| --- |
| ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags, struct sockaddr \*src\_addr, socklen\_t \*addrlen); |



### 非阻塞式I/O

应用进程执行系统调用之后，内核返回一个错误码。应用进程可以继续执行，但是需要不断的执行系统调用来获知 I/O 是否完成，这种方式称为轮询（polling）。

由于 CPU 要处理更多的系统调用，因此这种模型的 CPU 利用率比较低。

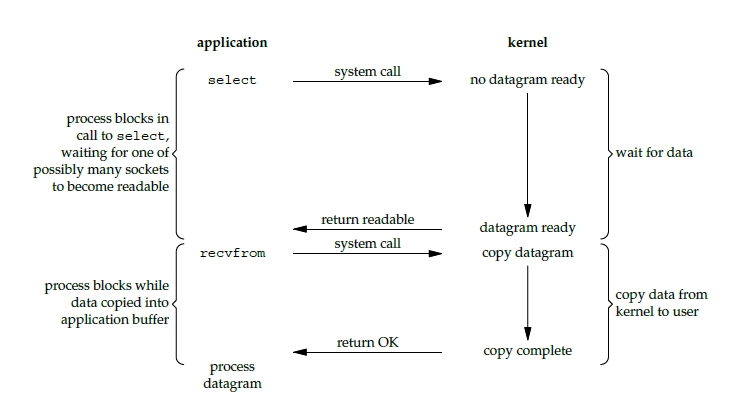


### I/O复用

使用 select 或者 poll 等待数据，并且可以等待多个套接字中的任何一个变为可读。这一过程会被阻塞，当某一个套接字可读时返回，之后再使用 recvfrom 把数据从内核复制到进程中。

它可以让单个进程具有处理多个 I/O 事件的能力。又被称为 Event Driven I/O，即事件驱动 I/O。

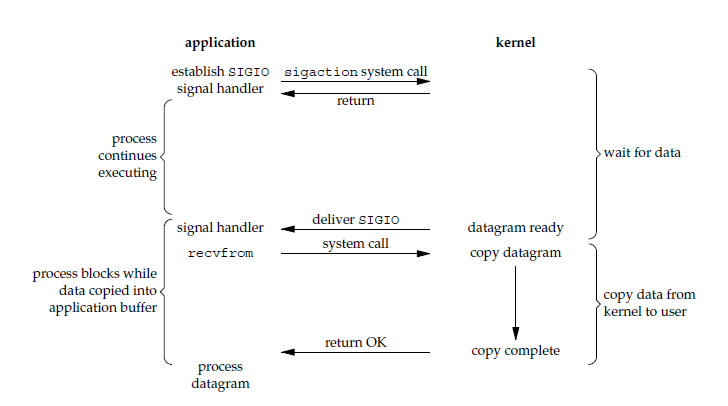
如果一个 Web 服务器没有 I/O 复用，那么每一个 Socket 连接都需要创建一个线程去处理。如果同时有几万个连接，那么就需要创建相同数量的线程。相比于多进程和多线程技术，I/O 复用不需要进程线程创建和切换的开销，系统开销更小。



### 信号驱动I/O

应用进程使用 sigaction 系统调用，内核立即返回，应用进程可以继续执行，也就是说等待数据阶段应用进程是非阻塞的。内核在数据到达时向应用进程发送 SIGIO 信号，应用进程收到之后在信号处理程序中调用 recvfrom 将数据从内核复制到应用进程中。

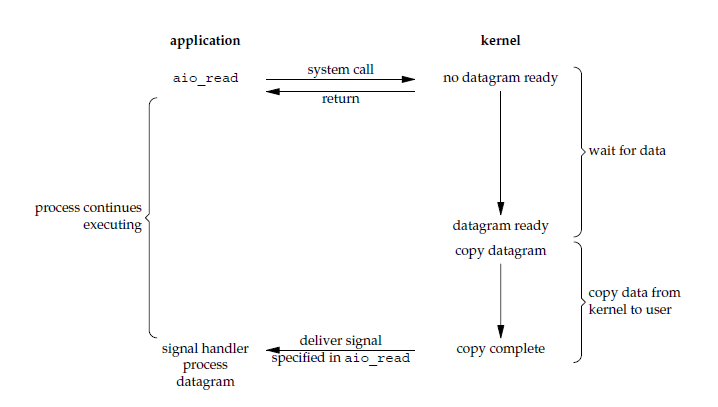
相比于非阻塞式 I/O 的轮询方式，信号驱动 I/O 的 CPU 利用率更高。



### 异步I/O

应用进程执行 aio\_read 系统调用会立即返回，应用进程可以继续执行，不会被阻塞，内核会在所有操作完成之后向应用进程发送信号。

异步 I/O 与信号驱动 I/O 的区别在于，异步 I/O 的信号是通知应用进程 I/O 完成，而信号驱动 I/O 的信号是通知应用进程可以开始 I/O。

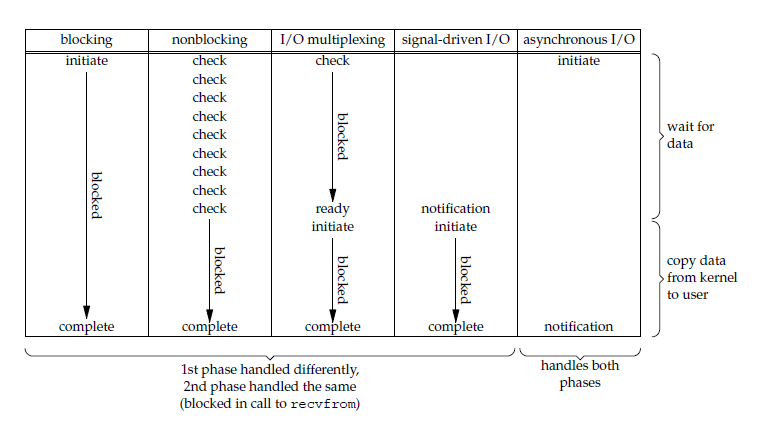


### 五大I/O模型比较

* 同步I/O：将数据从内核缓冲区复制到应用进程缓冲区的阶段，应用进程会阻塞。
* 异步I/O：不会阻塞

阻塞式 I/O、非阻塞式 I/O、I/O 复用和信号驱动 I/O 都是同步 I/O，它们的主要区别在第一个阶段。

非阻塞式 I/O 、信号驱动 I/O 和异步 I/O 在第一阶段不会阻塞。



## I/O复用

select/poll/epoll 都是 I/O 多路复用的具体实现，select 出现的最早，之后是 poll，再是 epoll。

### select

|  |
| --- |
| int select(int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout); |

有三种类型的描述符类型：readset、writeset、exceptset，分别对应读、写、异常条件的描述符集合。fd\_set 使用数组实现，数组大小使用 FD\_SETSIZE 定义。

timeout 为超时参数，调用 select 会一直阻塞直到有描述符的事件到达或者等待的时间超过 timeout。

成功调用返回结果大于 0，出错返回结果为 -1，超时返回结果为 0。

|  |
| --- |
| fd\_set fd\_in, fd\_out;  struct timeval tv;  // Reset the sets  FD\_ZERO( &fd\_in );  FD\_ZERO( &fd\_out );  // Monitor sock1 for input events  FD\_SET( sock1, &fd\_in );  // Monitor sock2 for output events  FD\_SET( sock2, &fd\_out );  // Find out which socket has the largest numeric value as select requires it  int largest\_sock = sock1 > sock2 ? sock1 : sock2;  // Wait up to 10 seconds  tv.tv\_sec = 10;  tv.tv\_usec = 0;  // Call the select  int ret = select( largest\_sock + 1, &fd\_in, &fd\_out, NULL, &tv );  // Check if select actually succeed  if ( ret == -1 )  // report error and abort  else if ( ret == 0 )  // timeout; no event detected  else  {  if ( FD\_ISSET( sock1, &fd\_in ) )  // input event on sock1  if ( FD\_ISSET( sock2, &fd\_out ) )  // output event on sock2  } |

### poll

|  |
| --- |
| int poll(struct pollfd \*fds, unsigned int nfds, int timeout); |

pollfd 使用链表实现。

|  |
| --- |
| // The structure for two events  struct pollfd fds[2];  // Monitor sock1 for input  fds[0].fd = sock1;  fds[0].events = POLLIN;  // Monitor sock2 for output  fds[1].fd = sock2;  fds[1].events = POLLOUT;  // Wait 10 seconds  int ret = poll( &fds, 2, 10000 );  // Check if poll actually succeed  if ( ret == -1 )  // report error and abort  else if ( ret == 0 )  // timeout; no event detected  else  {  // If we detect the event, zero it out so we can reuse the structure  if ( fds[0].revents & POLLIN )  fds[0].revents = 0;  // input event on sock1  if ( fds[1].revents & POLLOUT )  fds[1].revents = 0;  // output event on sock2  } |

### 比较

#### 功能

select和poll的功能基本相同，不过在一些实现细节上有所不同。

* select会修改描述符，而poll不会；
* select 的描述符类型使用数组实现，FD\_SETSIZE 大小默认为 1024，因此默认只能监听 1024 个描述符。如果要监听更多描述符的话，需要修改 FD\_SETSIZE 之后重新编译；而 poll 的描述符类型使用链表实现，没有描述符数量的限制；
* poll 提供了更多的事件类型，并且对描述符的重复利用上比 select 高。
* 如果一个线程对某个描述符调用了 select 或者 poll，另一个线程关闭了该描述符，会导致调用结果不确定。

#### 速度

select 和 poll 速度都比较慢。

* select 和 poll 每次调用都需要将全部描述符从应用进程缓冲区复制到内核缓冲区。
* select 和 poll 的返回结果中没有声明哪些描述符已经准备好，所以如果返回值大于 0 时，应用进程都需要使用轮询的方式来找到 I/O 完成的描述符。

#### 可移植性

几乎所有的系统都支持 select，但是只有比较新的系统支持 poll。

### epoll

|  |
| --- |
| int epoll\_create(int size);  int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)；  int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout); |

epoll\_ctl() 用于向内核注册新的描述符或者是改变某个文件描述符的状态。已注册的描述符在内核中会被维护在一棵红黑树上，通过回调函数内核会将 I/O 准备好的描述符加入到一个链表中管理，进程调用 epoll\_wait() 便可以得到事件完成的描述符。

从上面的描述可以看出，epoll 只需要将描述符从进程缓冲区向内核缓冲区拷贝一次，并且进程不需要通过轮询来获得事件完成的描述符。

epoll 仅适用于 Linux OS。

epoll 比 select 和 poll 更加灵活而且没有描述符数量限制。

epoll 对多线程编程更有友好，一个线程调用了 epoll\_wait() 另一个线程关闭了同一个描述符也不会产生像 select 和 poll 的不确定情况。

|  |
| --- |
| // Create the epoll descriptor. Only one is needed per app, and is used to monitor all sockets.  // The function argument is ignored (it was not before, but now it is), so put your favorite number here  int pollingfd = epoll\_create( 0xCAFE );  if ( pollingfd < 0 )  // report error  // Initialize the epoll structure in case more members are added in future  struct epoll\_event ev = { 0 };  // Associate the connection class instance with the event. You can associate anything  // you want, epoll does not use this information. We store a connection class pointer, pConnection1  ev.data.ptr = pConnection1;  // Monitor for input, and do not automatically rearm the descriptor after the event  ev.events = EPOLLIN | EPOLLONESHOT;  // Add the descriptor into the monitoring list. We can do it even if another thread is  // waiting in epoll\_wait - the descriptor will be properly added  if ( epoll\_ctl( epollfd, EPOLL\_CTL\_ADD, pConnection1->getSocket(), &ev ) != 0 )  // report error  // Wait for up to 20 events (assuming we have added maybe 200 sockets before that it may happen)  struct epoll\_event pevents[ 20 ];  // Wait for 10 seconds, and retrieve less than 20 epoll\_event and store them into epoll\_event array  int ready = epoll\_wait( pollingfd, pevents, 20, 10000 );  // Check if epoll actually succeed  if ( ret == -1 )  // report error and abort  else if ( ret == 0 )  // timeout; no event detected  else  {  // Check if any events detected  for ( int i = 0; i < ret; i++ )  {  if ( pevents[i].events & EPOLLIN )  {  // Get back our connection pointer  Connection \* c = (Connection\*) pevents[i].data.ptr;  c->handleReadEvent();  }  }  } |

### 工作模式

epoll 的描述符事件有两种触发模式：LT（level trigger）和 ET（edge trigger）。

#### LT模式

当 epoll\_wait() 检测到描述符事件到达时，将此事件通知进程，进程可以不立即处理该事件，下次调用 epoll\_wait() 会再次通知进程。是默认的一种模式，并且同时支持 Blocking 和 No-Blocking。

#### ET模式

和 LT 模式不同的是，通知之后进程必须立即处理事件，下次再调用 epoll\_wait() 时不会再得到事件到达的通知。

很大程度上减少了 epoll 事件被重复触发的次数，因此效率要比 LT 模式高。只支持 No-Blocking，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

### 应用场景

很容易产生一种错觉认为只要用 epoll 就可以了，select 和 poll 都已经过时了，其实它们都有各自的使用场景。

#### select应用场景

select 的 timeout 参数精度为 1ns，而 poll 和 epoll 为 1ms，因此 select 更加适用于实时性要求比较高的场景，比如核反应堆的控制。

select 可移植性更好，几乎被所有主流平台所支持。

#### poll应用场景

poll 没有最大描述符数量的限制，如果平台支持并且对实时性要求不高，应该使用 poll 而不是 select。

#### epoll应用场景

只需要运行在 Linux 平台上，有大量的描述符需要同时轮询，并且这些连接最好是长连接。

需要同时监控小于 1000 个描述符，就没有必要使用 epoll，因为这个应用场景下并不能体现 epoll 的优势。

需要监控的描述符状态变化多，而且都是非常短暂的，也没有必要使用 epoll。因为 epoll 中的所有描述符都存储在内核中，造成每次需要对描述符的状态改变都需要通过 epoll\_ctl() 进行系统调用，频繁系统调用降低效率。并且 epoll 的描述符存储在内核，不容易调试。