# Redis-list

## ****1.介绍****：

**Redis**中一个常用的数据结构就是**list**，其底层有**linkedList**、**zipList**和**quickList**这三种存储方式。

## ****2.常用命令:****

rpush,lpop,lpush,rpop,lrange,llen 等。

## ****3.应用场景:****

发布与订阅或者说消息队列、慢查询。

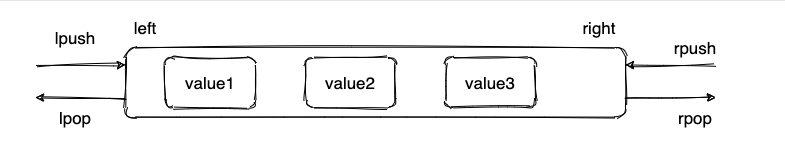
### 3.1****通过**rpush/lpop**实现队列：****

|  |
| --- |
| 127.0.0.1:6379> rpush myList value1 # 向 list 的头部（右边）添加元素  (integer) 1  127.0.0.1:6379> rpush myList value2 value3 # 向list的头部（最右边）添加多个元素  (integer) 3  127.0.0.1:6379> lpop myList # 将 list的尾部(最左边)元素取出"value1"  127.0.0.1:6379> lrange myList 0 1 # 查看对应下标的list列表， 0 为 start,1为 end  1) "value2"  2) "value3"  127.0.0.1:6379> lrange myList 0 -1 # 查看列表中的所有元素，-1表示倒数第一  1) "value2"  2) "value3" |

### ****3.2通过**rpush/rpop**实现栈：****

|  |
| --- |
| 127.0.0.1:6379> rpush myList2 value1 value2 value3  (integer) 3  127.0.0.1:6379> rpop myList2 # 将 list的头部(最右边)元素取出"value3" |

图解：



### 3.3****通过**lrange**查看对应下标范围的列表元素：****

|  |
| --- |
| 127.0.0.1:6379> rpush myList value1 value2 value3  (integer) 3  127.0.0.1:6379> lrange myList 0 1 # 查看对应下标的list列表， 0 为 start,1为 end  1) "value1"  2) "value2"  127.0.0.1:6379> lrange myList 0 -1 # 查看列表中的所有元素，-1表示倒数第一  1) "value1"  2) "value2"  3) "value3" |

因此：通过 lrange 命令，你可以基于 list 实现分页查询，性能非常高！

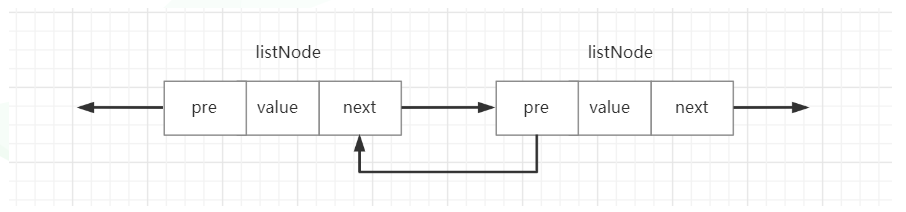
## 4.底层实现：

### 4.1.链表LinkedList

与**Java**中的**LinkedList**类似，**Redis**中的**linkedList**是一个双向链表，也是由一个个节点组成的。**Redis**中借助**C**语言实现的链表节点结构如下所示：

|  |
| --- |
| //定义链表节点的结构体  typedf struct listNode{  //前一个节点  struct listNode \*prev;  //后一个节点  struct listNode \*next;  //当前节点的值的指针  void \*value;  }listNode; |

**pre**指向前一个节点，**next**指针指向后一个节点，**value**保存着当前节点对应的数据对象。**listNode**的示意图如下所示：

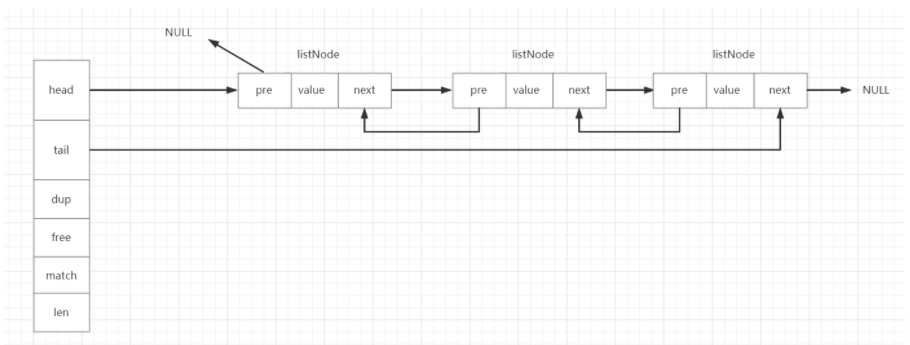


链表的结构如下：

|  |
| --- |
| typedf struct list{  //头指针  listNode \*head;  //尾指针  listNode \*tail;  //节点拷贝函数  void \*(\*dup)(void \*ptr);  //释放节点函数  void \*(\*free)(void \*ptr);  //判断两个节点是否相等的函数  int (\*match)(void \*ptr,void \*key);  //链表长度  unsigned long len;  } |

**head**指向链表的头节点，**tail**指向链表的尾节点，**dup**函数用于链表转移复制时对节点**value**拷贝的一个实现，一般情况下使用**等号**足以，但在某些特殊情况下可能会用到节点转移函数，默认可以给这个函数赋值**NULL**即表示使用等号进行节点转移。**free**函数用于释放一个节点所占用的内存空间，默认赋值**NULL**的话，即使用**Redis**自带的**zfree**函数进行内存空间释放。**match**函数是用来比较两个链表节点的**value**值是否相等，相等返回1，不等返回0。**len**表示这个链表共有多少个节点，这样就可以在**O(1)**的时间复杂度内获得链表的长度。

链表的结构如下所示：

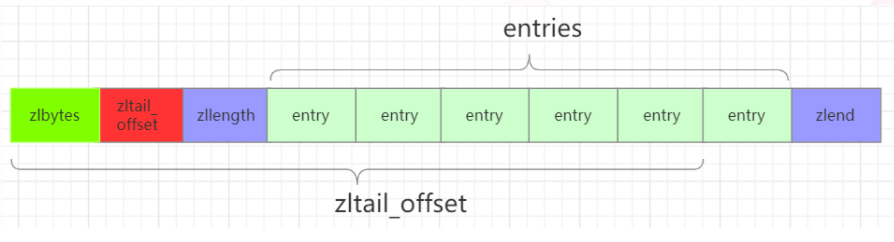


### 4.2.zipList

**Redis**的**zipList**结构如下所示：

|  |
| --- |
| typedf struct ziplist<T>{  //压缩列表占用字符数  int32 zlbytes;  //最后一个元素距离起始位置的偏移量，用于快速定位最后一个节点  int32 zltail\_offset;  //元素个数  int16 zllength;  //元素内容  T[] entries;  //结束位 0xFF  int8 zlend;  }ziplist |

**zipList**的结构如下所示：

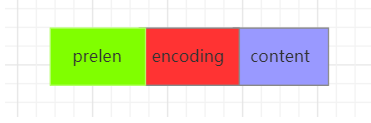


注意到**zltail\_offset**这个参数，有了这个参数就可以快速定位到最后一个**entry**节点的位置，然后开始倒序遍历，也就是说**zipList**支持双向遍历。

下面是**entry**的结构：

|  |
| --- |
| typede struct entry{  //前一个entry的长度  int<var> prelen;  //元素类型编码  int<var> encoding;  //元素内容  optional byte[] content;  }entry |

**prelen**保存的是前一个**entry**节点的长度，这样在倒序遍历时就可以通过这个参数定位到上一个**entry**的位置。**encoding**保存了**content**的编码类型。**content**则是保存的元素内容，它是**optional**类型的，表示这个字段是可选的。当**content**是很小的整数时，它会内联到**content**字段的尾部。**entry**结构的示意图如下所示：



思考一个问题，为什么有了**linkedList**还有设计一个**zipList**呢？就像**zipList**的名字一样，它是一个压缩列表，是为了节约内存而开发的。相比于**linkedList**，其少了**pre**和**next**两个指针。在**Redis**中，**pre**和**next**指针就要占用16个字节（64位系统的一个指针就是8个字节）。另外，**linkedList**的每个节点的内存都是单独分配，加剧内存的碎片化，影响内存的管理效率。与之相对的是，**zipList**是由连续的内存组成的，这样一来，由于内存是连续的，就减少了许多内存碎片和指针的内存占用，进而节约了内存。

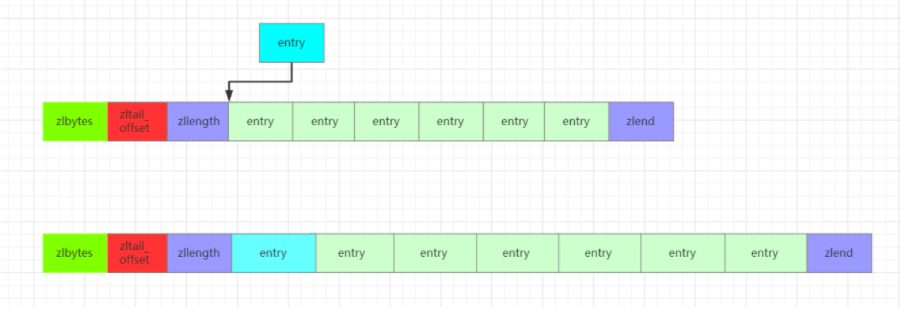
**zipList**遍历时，先根据**zlbytes**和**zltail\_offset**定位到最后一个**entry**的位置，然后再根据最后一个**entry**里的**prelen**时确定前一个**entry**的位置。

### 4.2.1.连锁更新

上面说到了，**entry**中有一个**prelen**字段，它的长度要么是1个字节，要么都是5个字节：

* 前一个节点的长度小于254个字节，则**prelen**长度为1字节；
* 前一个节点的长度大于254字节，则**prelen**长度为5字节；

假设现在有一组压缩列表，长度都在250~253字节之间，突然新增一个**entry**节点，这个**entry**节点长度大于等于254字节。由于新的**entry**节点大于等于254字节，这个**entry**节点的**prelen**为5个字节，随后会导致其余的所有**entry**节点的**prelen**增大为5字节。



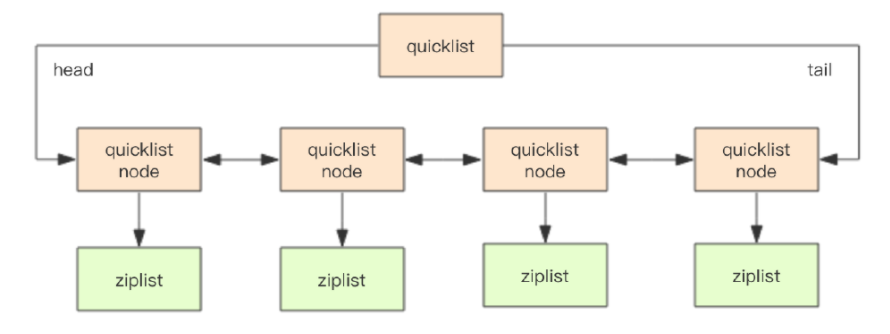
同样地，删除操作也会导致出现**连锁更新**这种情况，假设在某一时刻，插入一个长度大于等于254个字节的**entry**节点，同时删除其后面的一个长度小于254个字节的**entry**节点，由于小于254的**entry**节点的删除，大于等于254个字节的**entry**节点将会与后面小于254个字节的**entry**节点相连，此时就与新增一个长度大于等于254个字节的**entry**节点时的情况一样，将会发生连续更新。发生连续更新时，**Redis**需要不断地对压缩列表进行**内存分配工作**，直到结束。

### 4.3.linkedList与zipList 对比

* 当列表对象中元素的长度较小或者数量较少时，通常采用**zipList**来存储；当列表中元素的长度较大或者数量比较多的时候，则会转而使用双向链表**linkedList**来存储。
* 双向链表**linkedList**便于在表的两端进行**push**和**pop**操作，在插入节点上复杂度很低，但是它的内存开销比较大。首先，它在每个节点上除了要保存数据之外，还有额外保存两个指针；其次，双向链表的各个节点都是单独的内存块，地址不连续，容易形成内存碎片。
* **zipList**存储在一块连续的内存上，所以存储效率很高。但是它不利于修改操作，插入和删除操作需要频繁地申请和释放内存。特别是当**zipList**长度很长时，一次**realloc**可能会导致大量的数据拷贝。

### 4.4.quickList

在**Redis**3.2版本之后，**list**的底层实现方式又多了一种，**quickList**。**qucikList**是由**zipList**和双向链表**linkedList**组成的混合体。它将**linkedList**按段切分，每一段使用**zipList**来紧凑存储，多个**zipList**之间使用双向指针串接起来。示意图如下所示：



节点**quickListNode**的定义如下：

|  |
| --- |
| typedf struct quicklistNode{  //前一个节点  quicklistNode\* prev;  //后一个节点  quicklistNode\* next;  //压缩列表  ziplist\* zl;  //ziplist大小  int32 size;  //ziplist 中元素数量  int16 count;  //编码形式 存储 ziplist 还是进行 LZF 压缩储存的zipList  int2 encoding; ...  }quickListNode |

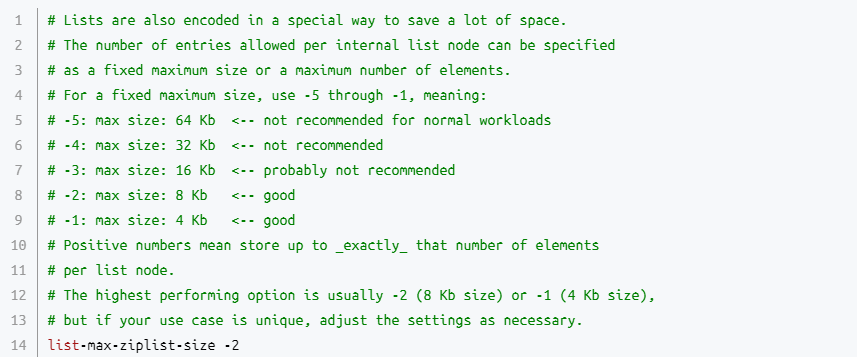
**quickList**的定义如下所示：

|  |
| --- |
| typedf struct quicklist{  //指向头结点  quicklistNode\* head;  //指向尾节点  quicklistNode\* tail;  //元素总数  long count;  //quicklistNode节点的个数  int nodes;  //压缩算法深度  int compressDepth; ...  }quickList |

上述代码简单地表示了**quickList**的大致结构，为了进一步节约空间，**Redis**还会对**zipList**进行压缩存储，使用**LZF**算法进行压缩，可以选择压缩深度。

### 4.4.1.每个zipList可以存储多少个元素

想要了解这个问题，就得打开**redis.conf**文件了。在**DVANCED CONFIG**下面有着清晰的记载。



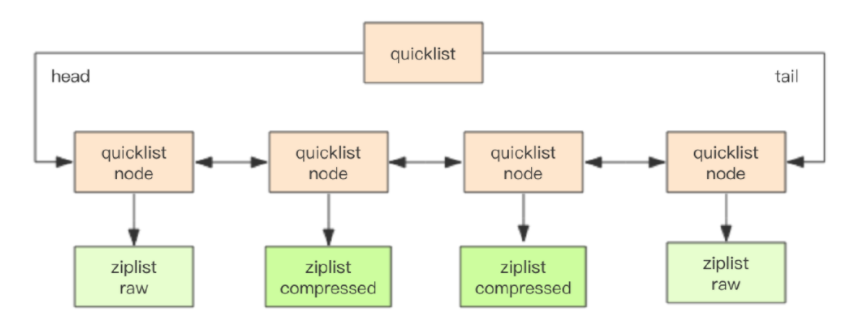
**quickList**内部默认单个**zipList**长度为**8k**字节，即**list-max-ziplist-size**的值设置为**-2**，超出了这个阈值，就会重新生成一个**zipList**来存储数据。根据注释可知，性能最好的时候就是就是**list-max-ziplist-size**为**-1**和**-2**，即分别是**4kb和8kb**的时候，当然，这个值也可以被设置为正数，当**list-max-ziplist-szie**为**正数n**时，表示每个**quickList**节点上的**zipList**最多包含**n个**数据项。

### 4.4.2.压缩深度

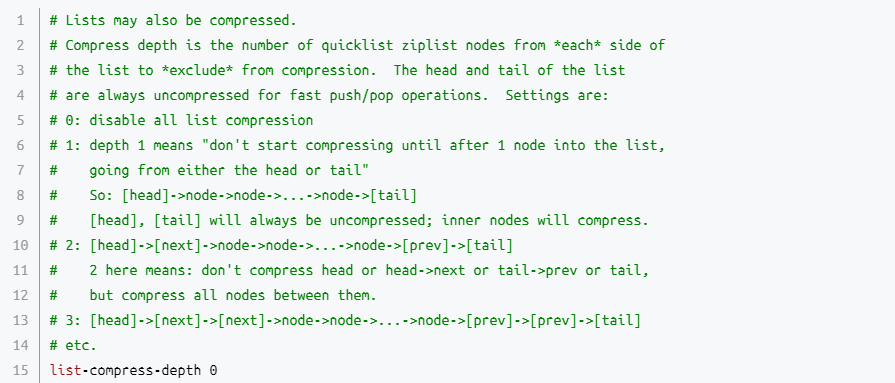
上面提到过，**quickList**中可以使用压缩算法对**zipList**进行进一步的压缩，这个算法就是[LZF算法](https://blog.csdn.net/u012319493/article/details/83653860" \t "https://www.cnblogs.com/reecelin/p/_blank)，这是一种无损压缩算法，具体可以参考这里。使用压缩算法对**zipList**进行压缩后，**zipList**的结构如下所示：

|  |
| --- |
| typedf struct ziplist\_compressed{  //元素个数  int32 size;  //元素内容  byte[] compressed\_data  } |

此时**quickList**的示意图如下所示：



当然，在**redis.conf**文件中的**DVANCED CONFIG**下面也可以对压缩深度进行配置。

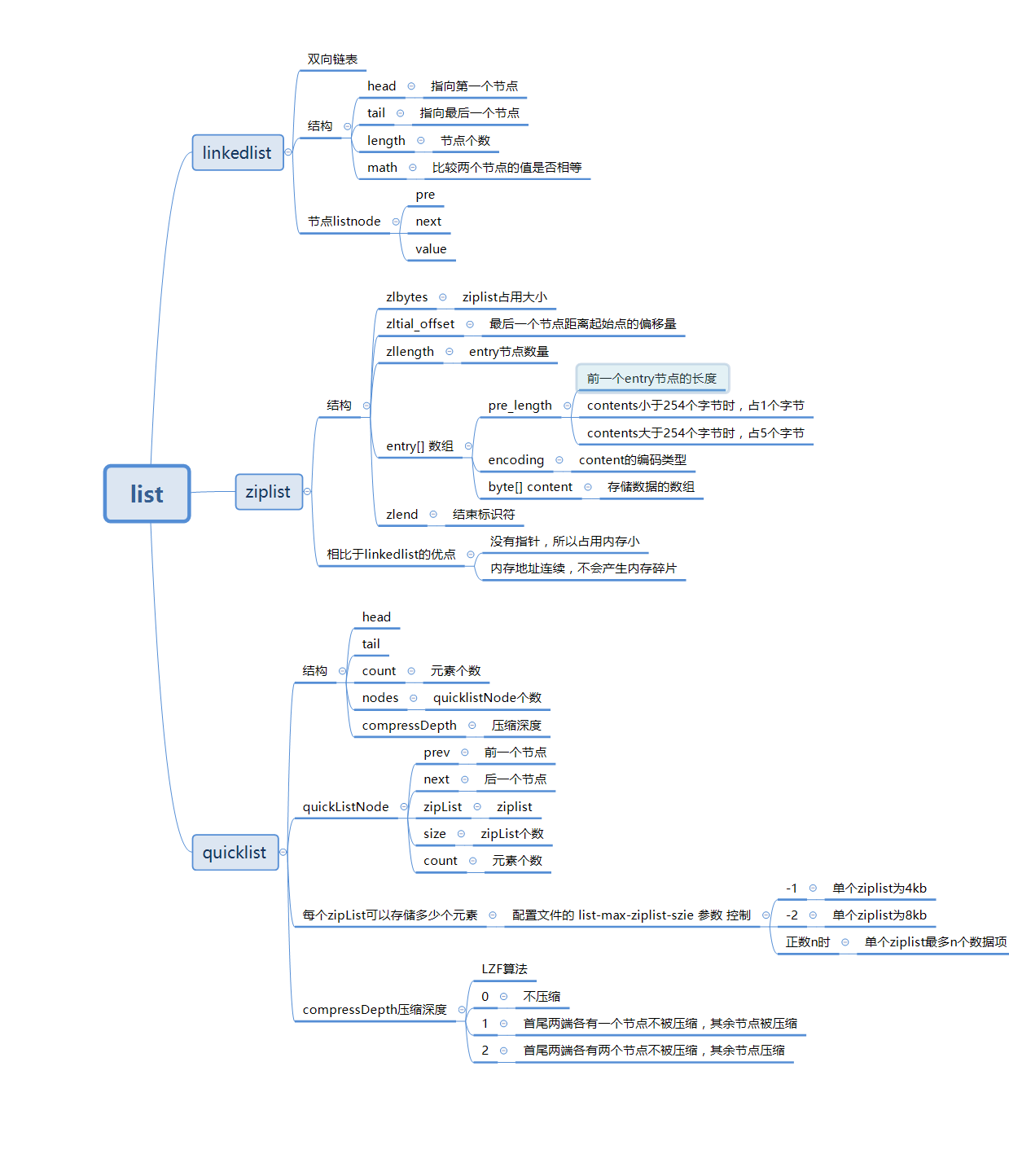


**list-compress-depth**这个参数表示**一个quickList两端不被压缩的节点个数。**需要注意的是，这里的节点个数是指**quicklist**双向链表的节点个数，而不是指**ziplis**t里面的数据项个数。实际上，一个**quicklist**节点上的**ziplist**，如果被压缩，就是整体被压缩的。

* **quickList**默认的压缩深度为**0**，也就是不开启压缩
* 当**list-compress-depth**为1，表示**quickList**的两端各有1个节点不进行压缩，中间结点进行压缩；
* 当**list-compress-depth**为2，表示**quickList**的首尾2个节点不进行压缩，中间结点进行压缩；
* 以此类推

从上面可以看出，对于**quickList**来说，其首尾两个节点永远不会被压缩。

## 5.总结：



## 6.文章分享：

[https://juejin.im/post/5de3e841f265da05d03826ca#heading-6](https://juejin.im/post/5de3e841f265da05d03826ca" \l "heading-6" \t "https://www.cnblogs.com/reecelin/p/_blank)

[https://juejin.im/post/5cfd1bd6f265da1b7b317e07#heading-3](https://juejin.im/post/5cfd1bd6f265da1b7b317e07" \l "heading-3" \t "https://www.cnblogs.com/reecelin/p/_blank)

[https://juejin.im/post/5df9df506fb9a0160b6380f5](https://juejin.im/post/5df9df506fb9a0160b6380f5" \t "https://www.cnblogs.com/reecelin/p/_blank)