qinm的专栏

一步一个脚印。。

个人资料

qinm

访问： 30989次
积分： 1547
等级： BLOG 4
排名： 第19214名

原创： 118篇
转载： 21篇
译文： 0篇
评论： 4条

leveldb之SSTable

2015-05-10 16:30 182人阅读 评论(0) 收藏 举报

分类： leveldb（13）

转载：http://blog.csdn.net/tankles/article/details/7663905

SSTable是Bigtable中至关重要的一块，对于LevelDB来说也是如此，对LevelDB的SSTable实现细节的了解也有助于了解Bigtable中一些实现细节。

1、SSTable的布局

本节主要介绍SSTable某个文件的物理布局和逻辑布局结构，这对了解LevelDB的运行过程很有帮助。

LevelDB不同层级都有一个或多个SSTable文件（以后缀.sst为特征），所有.sst文件内部布局都是一样的。上节介绍Log文件是物理分块的，SSTable也一样会将文件划分为固定大小的物理存储块Block，但是两者逻辑布局大不相同，根本原因是：Log文件中的记录是Key无序的，即先后记录的key大小没有明确大小关系，而.sst文件内部则是根据记录的Key由小到大排列的，从下面介绍的SSTable布局可以体会到Key有序是为何如此设计.sst文件结构的关键。

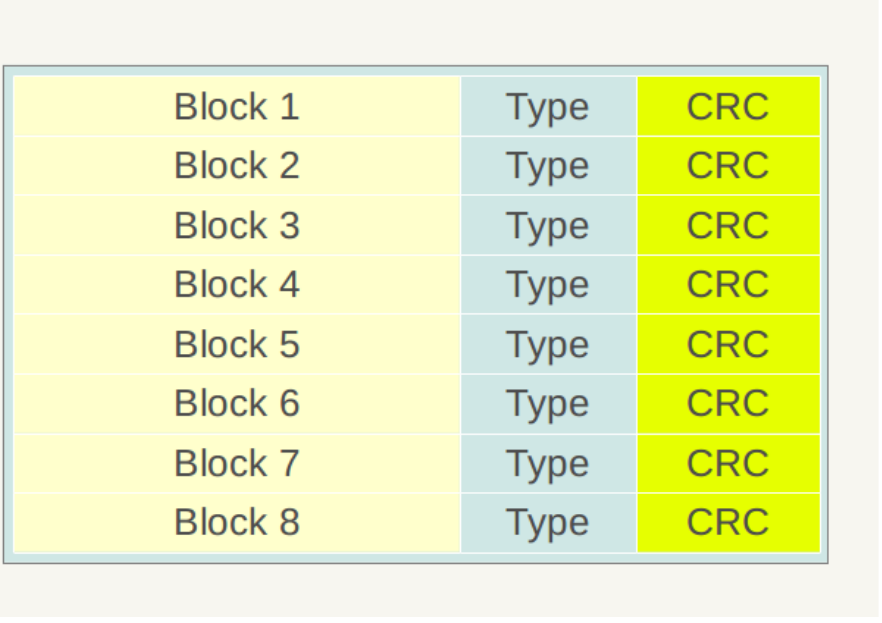


图1 .sst文件的分块结构

图1展示了一个.sst文件的物理划分结构，同Log文件一样，也是划分为固定大小的存储块，每个Block分为三个部分，包括Block、Type和CRC。Block为数据存储区，Type区用于标识Block中数据是否采用了数据压缩算法（Snappy压缩或者无压缩两种），CRC部分则是Block数据校验码，用于判别数据是否在生成和传输中出错。

以上是.sst的物理布局，下面介绍.sst文件的逻辑布局，所谓逻辑布局，就是说尽管大家都是物理块，但是每一块存储什么内容，内部又有什么结构等。图4.2展示了.sst文件的内部逻辑解释。

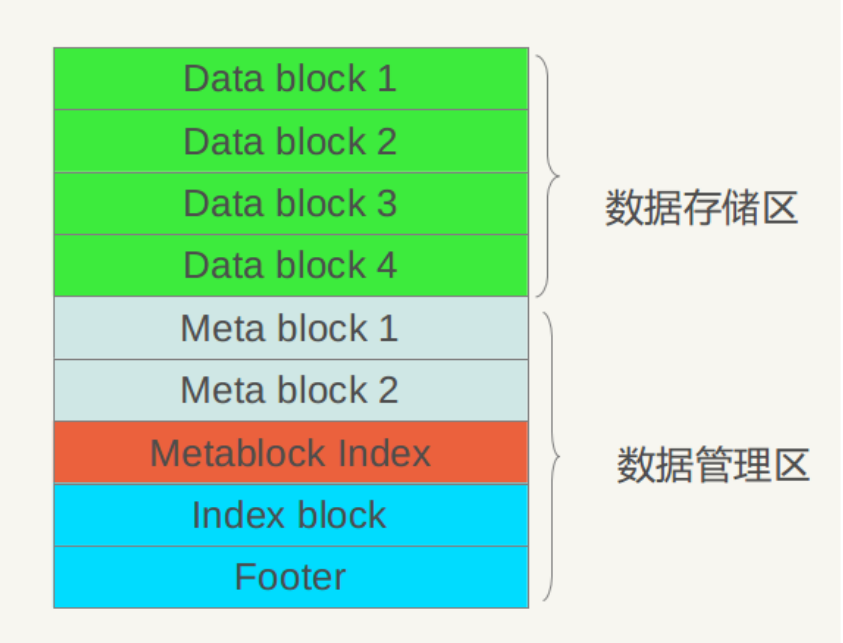


图2 逻辑布局

从图2可以看出，从大的方面，可以将.sst文件划分为数据存储区和数据管理区，数据存储区存放实际的Key:Value数据，数据管理区则提供一些索引指针等管理数据，目的是更快速便捷的查找相应的记录。两个区域都是在上述的分块基础上的，就是说文件的前面若干块实际存储KV数据，后面数据管理区存储管理数据。管理数据又分为四种不同类型：紫色的Meta Block，红色的MetaBlock Index和蓝色的Index block以及一个文件尾部块Footer。

2、SSTable的数据管理区

LevelDB 1.2版对于Meta Block尚无实际使用，只是保留了一个接口，估计会在后续版本中加入内容，下面我们看看Index block和文件尾部Footer的内部结构。

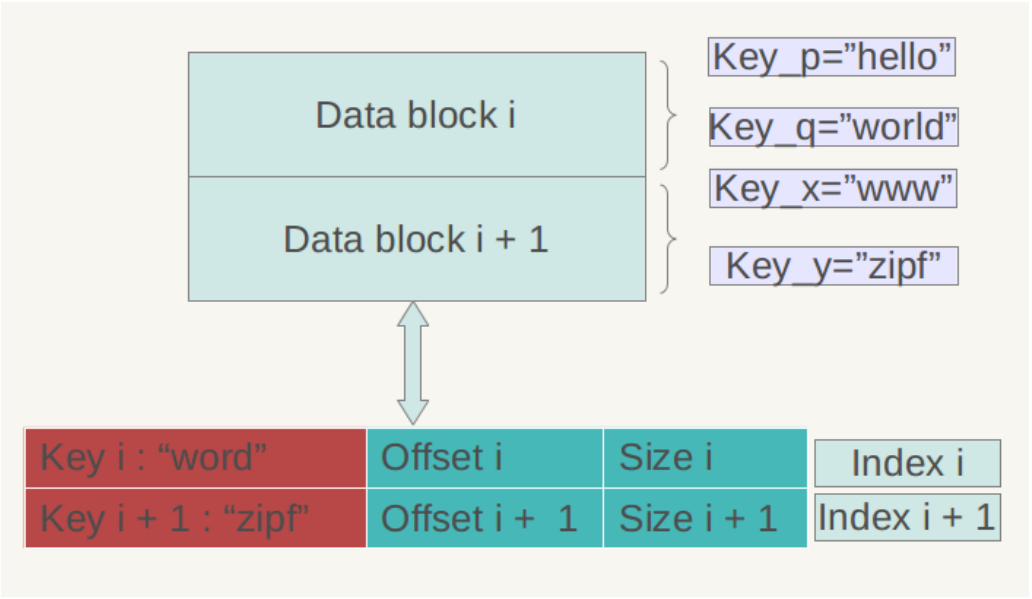


图3 Index block结构

图3是Index block的内部结构示意图。再次强调一下，Data Block内的KV记录是按照Key由小到大排列的，Index block的每条记录是对某个Data Block建立的索引信息，每条索引信息包含三个内容：Data Block中key上限值(不一定是最大key)、Data Block在.sst文件的偏移和大小，以图3所示的数据块i的索引Index i来说：红色部分的第一个字段记载大于等于数据块i中最大的Key值的那个Key，第二个字段指出数据块i在.sst文件中的起始位置，第三个字段指出Data Block i的大小（有时候是有数据压缩的）。后面两个字段好

理解，是用于定位数据块在文件中的位置的，第一个字段需要详细解释一下，在索引里保存的这个Key值未必一定是某条记录的Key,以图3的例子来说，假设数据块i 的最小Key=“samecity”，最大Key=“the best”;数据块i+1的最小Key=“the fox”,最大Key=“zoo”,那么对于数据块i的索引Index i来说，其第一个字段记载大于等于数据块i的最大Key(“the best”)，同时要小于数据块i+1的最小Key(“the fox”)，所以例子中Index i的第一个字段是：“the c”，这个是满足要求的；而Index i+1的第一个字段则是“zoo”，即数据块i+1的最大Key。

文件末尾Footer块的内部结构见图4，metaindex_handle指出了metaindex block的起始位置和大小；inex_handle指出了index Block的起始地址和大小；这两个字段可以理解为索引的索引，是为了正确读出索引值而设立的，后面跟着一个填充区和魔数（0xdb4775248b80fb57）。

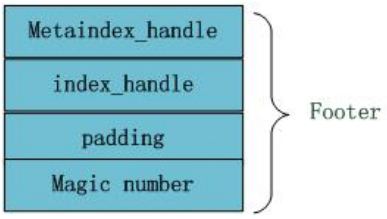


图4 Footer

3、SSTable的数据区

上面主要介绍的是数据管理区的内部结构，下面我们看看数据区的一个Block的数据部分内部是如何布局的，图5是其内部布局示意图。

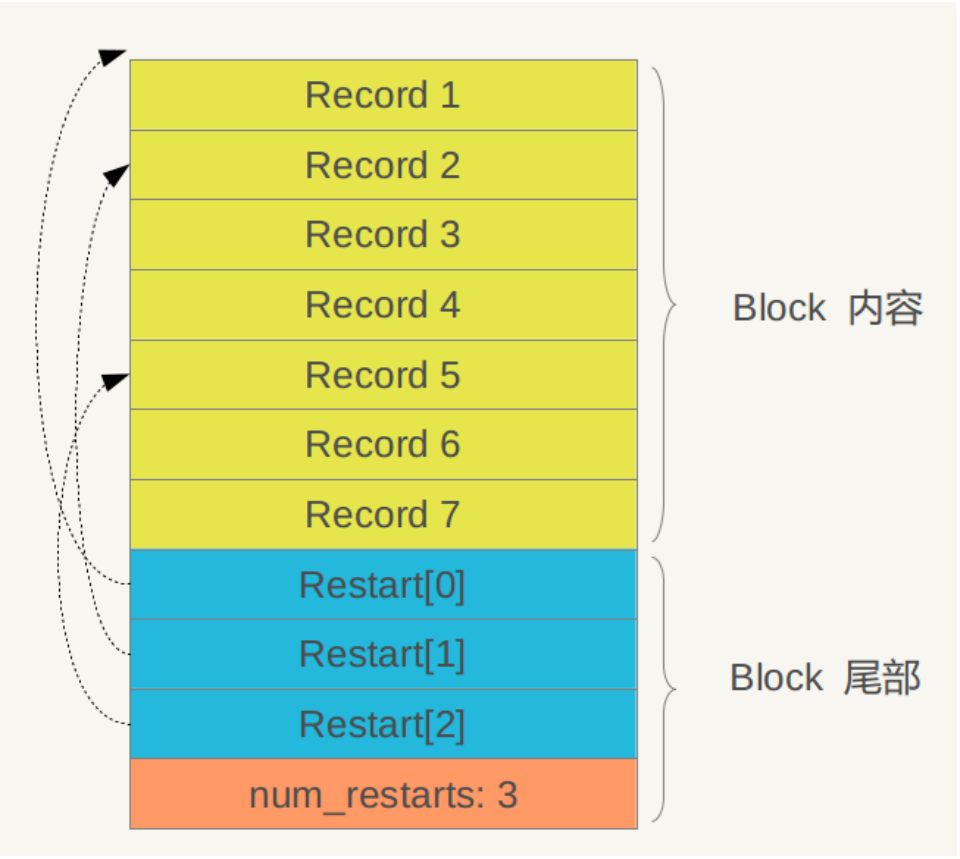


图5 Data Block内部结构

从图中可以看出，其内部也分为两个部分，前面是一个个KV记录，其顺序是根据Key值由小到大排列的，在Block尾部有一个数组，记录Block中每一个记录重启点相对于buffer_的偏移，最后用一个uint32_t数值记录重启点的总个数，通过restart_.size()得到。

由上图可以清晰的知道每一个构建完成的Block的总大小为：

```cpp
size_t BlockBuilder::CurrentSizeEstimate() const {
  return (buffer_.size() +                        // 所有K-V记录占用的空间
          restarts_.size() * sizeof(uint32_t) +   // 所有重启点占用的空间
          sizeof(uint32_t));                      //重启点个数：restart_.size()
}
```

缩

提取共同前缀，即进行前缀压缩来减少存储空间。

里的KV记录是按照Key大小有序的，这样的话，相邻的两条记录很可能Key部分存在重叠，比如key i="the car"，Key i+1="the color",那么两者存在重叠部分"the c"，为了减少Key的存储量，Key i+1可以只存储和上一条Key不同的部分"olor"，两者的共同部分从Key i中可以获得。

3.2重启点

重启点"的意思是：在这条记录开始，不再采取只记载不同的Key部分，而是重新记录完整的Key值，[illegible]是一个重启点，那么Key里面会完整存储"the color"，而不是采用简略的"olor"方式。使用Block的最大目的是减少存储空间，减小访问开销，但是如果记录条数比较多，随机访问一条记录，需要从头开始一直解析才行，这样也产生很大的开销，所以设置了多个重启点。重启点通过option中的block_restart_intervalBlock参数进行配置，默认值为16，即每隔16条记录进行一次重启。在Block尾部的一个数组中存储着每一个重启点的位置。

Record i	key共享长度	key非共享长度	value长度	key非共享内容	value内容
Record i+1	key共享长度	key非共享长度	value长度	key非共享内容	value内容

图6 记录格式

在Block内容区，每个KV记录的内部结构是怎样的？图6给出了其详细结构，每个记录包含5个字段：key共享长度，key非共享长度，value长度，key非共享内容，value内容。比如上面的"the car"和"the color"记录，key共享长度5；key非共享长度是4；而key非共享内容则实际存储"olor"；value长度及内容分别指出Key:Value中Value的长度和存储实际的Value值。

4、BlockBuilder

在leveldb中是通过BlockBuilder来构建每一个block的

4.1BlockBuiler类的基本结构

BlockBuilder类的基本结构如下：

```cpp
class BlockBuilder {
 public:
  explicit BlockBuilder(const Options* options);
  void Reset();
  void Add(const Slice& key, const Slice& value);// REQUIRES: key is larger than any previously a
  Slice Finish();// Finish building the block and return a slice that refers to the block content
  size_t CurrentSizeEstimate() const; // Returns an estimate of the current (uncompressed) size o
  }
 private:
  const Options*        options_;
  std::string           buffer_;      // Destination buffer
  std::vector<uint32_t> restarts_;    // Restart points
  int                   counter_;     // Number of entries emitted since restart
  bool                  finished_;    // Has Finish() been called?
  std::string           last_key_;

  // No copying allowed
  BlockBuilder(const BlockBuilder&);
  void operator=(const BlockBuilder&);
};
```

buffer_中存储着每一条记录，restarts_数组中存储着每一个重启点的位置，counter_记录着每一次restart后的记录个数，当到达block_restart_intervalBlock个后重启记录。

4.2 BlockBuilder::Add(key, value)

通过Add()函数向Block中添加一条记录：

```cpp
void BlockBuilder::Add(const Slice& key, const Slice& value) {
  Slice last_key_piece(last_key_);//上一条记录的key值
  assert(buffer_.empty() // 为空，没有记录
         || options_->comparator->Compare(key, last_key_piece) > 0);//或者key比上一条记录的key值大，保证按key从小到达排列
  size_t shared = 0;
  if (counter_ < options_->block_restart_interval) {//判断是否需要重启，然后获得key共享长度
    const size_t min_length = std::min(last_key_piece.size(), key.size());
    while ((shared < min_length) && (last_key_piece[shared] == key[shared])) {
      shared++;
    }
  } else {
    restarts_.push_back(buffer_.size());//若需要重启，则记录重启点
    counter_ = 0;
  }
  const size_t non_shared = key.size() - shared;//key的非共享长度，重启时即为key.size()
  //填充每一条记录的前3部分：key共享长度、非共享长度和value.size()
  PutVarint32(&buffer_, shared);
  PutVarint32(&buffer_, non_shared);
  PutVarint32(&buffer_, value.size());

  buffer_.append(key.data() + shared, non_shared);//将key的非共享部分写入buffer_中
  buffer_.append(value.data(), value.size());//将value写入buffer_中，这样一条记录就完整的写入到buffer_中了

  // 然后更新变量信息
  last_key_.resize(shared);
  last_key_.append(key.data() + shared, non_shared);
  counter_++;
}
```

4.3BlockBuilder::Finish()

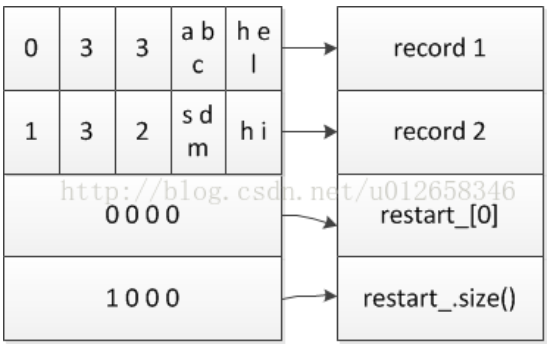
调用Finish()完成一个block的构建：

```cpp
Slice BlockBuilder::Finish() {
  for (size_t i = 0; i < restarts_.size(); i++) {
    PutFixed32(&buffer_, restarts_[i]);//将每一个重启点写入到buffer_中
  }
  PutFixed32(&buffer_, restarts_.size());//将重启点的总个数写入到buffer_中
  finished_ = true;
  return Slice(buffer_);//将buffer_转换为Slice结构并返回
}
```

4.4示例

```cpp
leveldb::Slice key("abc");
leveldb::Slice value("hel");
db->Add(key,value);
leveldb::Slice key1("asdm");
leveldb::Slice value1("hi");
db->Add(key1,value1);
leveldb::Slice res=db->Finish();
size_t size=db->CurrentSizeEstimate();
```

当插入上面两条记录，然后调用Finish()得到的结果如下：



构建完成后的Block占用的空间为：

```cpp
size=(buffer_.size() +           // 17
          restarts_.size() * sizeof(uint32_t) +   //1*4
          sizeof(uint32_t));  //4
 =25字节
```

5、Block

leveldb通过BlockBuilder来构建Block，用Block类来描述一个Block，并通过Block::Iter来对每一条记录进行操作的。

Block类的基本结构如下：

```cpp
class Block {
 public:
  explicit Block(const BlockContents& contents);
  ~Block();
  size_t size() const { return size_; }
  Iterator* NewIterator(const Comparator* comparator);
 private:
  uint32_t NumRestarts() const;
  const char* data_;
  size_t size_;
  uint32_t restart_offset_;     // restart_数组在缓冲区data中的偏移
  bool owned_;                  // Block owns data_[]

  // No copying allowed
  Block(const Block&);
  void operator=(const Block&);

  class Iter;
};
```

由上可知，Block是通过一个特定的BlockContents结构来初始化的，BlockContents结构如下：

```cpp
struct BlockContents {
  Slice data;           // Actual contents of data
  bool cachable;        // True iff data can be cached
  bool heap_allocated;  // True iff caller should delete[] data.data()
};
```

由上面对BlockBuilder的分析可知，通过BlockBuilder::Add()向一个Block中添加记录，通过Blockbuilder::Finish()完成一个Block的构建，并返回一个Slice结构，返回的结构如上面的图5所示。

因此将Finish()返回的Slice结构作为BlockContents的内容，然后传递给Block类，用来描述一个Block

5.1Block::Block()

```cpp
```

```
Block::Block(const BlockContents& contents)
    : data_(contents.data.data()),//slice.data()，将Slice的内容赋给char *data_
      size_(contents.data.size()),//slice.size()，缓冲区总大小
      owned_(contents.heap_allocated) {//是否是堆分配的，关系到释放时是否需要调用delete销毁掉
  if (size_ < sizeof(uint32_t)) {//size_不可能小于4字节，即使是空的也有4字节的restart.size()
    size_ = 0;  // Error marker
  } else {
    size_t max_restarts_allowed = (size_-sizeof(uint32_t)) / sizeof(uint32_t);//重启点的最大个数
    if (NumRestarts() > max_restarts_allowed) {
      size_ = 0;
    } else {
      restart_offset_ = size_ - (1 + NumRestarts()) * sizeof(uint32_t);
    }
  }
}
```

重启点的个数

```
[cpp]
inline uint32_t Block::NumRestarts() const {
  assert(size_ >= sizeof(uint32_t));
  return DecodeFixed32(data_ + size_ - sizeof(uint32_t));//表示的是最后4字节的内容
}
```

由上面的Block结构图可知，Block的最后4字节为restart_.size()，即重启点的总个数

这样就可以确定一个Block的结构了，然后通过Block::Iter来解析每一条记录

5.2Block::Iter::Seek()

Seek()首先对所有重启点进行二分查找，直到找到target对应的重启点，然后从重启点开始，线性查找每一条记录

```
[cpp]
virtual void Seek(const Slice& target) {
  uint32_t left = 0;
  uint32_t right = num_restarts_ - 1;
  while (left < right) {
    uint32_t mid = (left + right + 1) / 2;//采用二分法查找，找到target对应的记录对应的重启点
    uint32_t region_offset = GetRestartPoint(mid);//找到中间重启点在数据区的偏移，记录在restart数组中
    uint32_t shared, non_shared, value_length;
    const char* key_ptr = DecodeEntry(data_ + region_offset,
                                      data_ + restarts_,
                                      &shared, &non_shared, &value_length);//找到中间重启点的第一个key值
    Slice mid_key(key_ptr, non_shared);
    if (Compare(mid_key, target) < 0) {//由于每个重启点后的Key值是从小到大进行排列的，因此mid_key<target时，说明在后半部分
      left = mid;
    } else {//否则mid_key>=target，则应在前半部分
      right = mid - 1;
    }
  }
  //然后从找到的重启点开始，线性查找每一个key值，key<=target,当key>=target时查找结束
  SeekToRestartPoint(left);
  while (true) {
    if (!ParseNextKey()) {
      return;
    }
    if (Compare(key_, target) >= 0) {
      return;
    }
  }
}
```

顶 0 踩 0

上一篇 leveldb之cache

下一篇 leveldb之SSTable的创建与访问

我的同类文章

leveldb（13）

- leveldb之DBImpl 2015-05-18 阅读 266
- leveldb之文件 2015-05-15 阅读 276
- leveldb之SSTable的创建与… 2015-05-11 阅读 288
- leveldb之cache 2015-05-04 阅读 221
- leveldb之WriteBatch 2015-04-28 阅读 208
- leveldb之Compaction操作… 2015-05-17 阅读 409
- leveldb之Version相关数据… 2015-05-15 阅读 209
- leveldb之示意图 2015-05-12 阅读 223
- leveldb之SkipList的简单实现 2015-04-30 阅读 165
- leveldb之Arena 2015-04-28 阅读 184

更多文章

猜你在找

顾荣：开源大数据存储系统Alluxio（原Tachyon）的原 levelDB源码分析-SSTablesst文件构建与读取
360度解析亚马逊AWS数据存储服务 leveldb学习sstable1
Excel报表管理利器 LevelDB源码剖析之SSTable_sstable文件的创建
iOS开发高级专题—数据存储 详解SSTable结构和LSMTree索引
CentOS7 Linux系统管理实战视频课程 levedb之SSTable

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap