# 命令行参数解析——gflags

google开源的gflags是一套命令行参数解析工具，比getopt功能更强大，使用起来更加方便，gflags还支持从环境变量、配置文件读取参数（可用gflags代替配置文件）。本文简单介绍gflags的使用，内容主要译自 <http://gflags.googlecode.com/svn/trunk/doc/gflags.html> 。

定义参数   
使用flags需要包含头文件  #include   
gflags主要支持的参数类型包括bool，int32, int64, uint64, double, string等，定义参数通过DEFINE\_type宏实现，如下所示，分别定义了一个bool和一个string类型的参数，该宏的三个参数含义分别为命令行参数名，参数默认值，以及参数的帮助信息。

DEFINE\_bool(big\_menu, true, "Include 'advanced' options in the menu listing");   
DEFINE\_string(languages, "english,french,german",   
                 "comma-separated list of languages to offer in the 'lang' menu");   
gflag不支持列表，用户通过灵活借助string参数实现，比如上述的languages参数，可以类型为string，但可看作是以逗号分割的参数列表。

访问参数   
当参数被定义后，通过FLAGS\_name就可访问到对应的参数，比如上述定义的big\_menu、languages两个参数就可通过FLAGS\_big\_menu、FLAGS\_languages访问。

if (FLAGS\_languages.find("english") != string::npos)   
     HandleEnglish();

以上的访问方式，仅在参数定义和访问在同一个文件（或是通过头文件包含）时，FLAGS\_name才能访问到参数，如果要访问其他文件里定义的参数，则需要使用DECLARE\_type。比如在foo.cc中DEFINE\_string(color, "red", "the color you want to use"); 这是如果你需要在foo\_test.cc中使用color这个参数，你需要加入DECLARE\_string(color, "red", "the color you want to use");

参数检查   
定义参数后，可以给参数注册一个检查函数（validator），当从命令行指定参数或通过SetCommandLineOption()指定参数时，检查函数就会被调用，两个参数分别为命令行参数名，以及设置的参数值。   
static bool ValidatePort(const char\* flagname, int32 value) {   
   if (value > 0 && value < 32768)   // value is ok   
     return true;   
   printf("Invalid value for --%s: %d\n", flagname, (int)value);   
   return false;   
}   
DEFINE\_int32(port, 0, "What port to listen on");   
static const bool port\_dummy = RegisterFlagValidator(&FLAGS\_port, &ValidatePort);   
建议在定义参数后，立即注册检查函数。RegisterFlagValidator()在检查函数注册成功时返回true；如果参数已经注册了检查函数，或者检查函数类型不匹配，返回false。

初始化参数   
在引用程序的main()里通过 google::ParseCommandLineFlags(&argc, &argv, true); 即完成对gflags参数的初始，其中第三个参数为remove\_flag，如果为true，gflags会移除parse过的参数，否则gflags就会保留这些参数，但可能会对参数顺序进行调整。 比如 "/bin/foo" "arg1" "-q" "arg2"  会被调整为 "/bin/foo", "-q", "arg1", "arg2"，这样更好理解。

在命令行指定参数   
比如要在命令行指定languages参数的值，可通过如下4种方式，int32, int64等类型与string类似。

app\_containing\_foo --languages="chinese,japanese,korean"

app\_containing\_foo -languages="chinese,japanese,korean"

app\_containing\_foo --languages "chinese,japanese,korean"

app\_containing\_foo -languages "chinese,japanese,korean"

对于bool类型，则可通过如下几种方式指定参数

app\_containing\_foo --big\_menu

app\_containing\_foo --nobig\_menu

app\_containing\_foo --big\_menu=true

app\_containing\_foo --big\_menu=false

特殊参数

--help 打印定义过的所有参数的帮助信息

--version 打印版本信息 通过google::SetVersionString()指定

--nodefok  但命令行中出现没有定义的参数时，并不退出（error-exit）

--fromenv 从环境变量读取参数值 --fromenv=foo,bar表明要从环境变量读取foo，bar两个参数的值。通过export FLAGS\_foo=xxx; export FLAGS\_bar=yyy 程序就可读到foo，bar的值分别为xxx，yyy。

--tryfromenv 与--fromenv类似，当参数的没有在环境变量定义时，不退出（fatal-exit）

--flagfile 从文件读取参数值，--flagfile=my.conf表明要从my.conf文件读取参数的值。在配置文件中指定参数值与在命令行方式类似，另外在flagfile里可进一步通过--flagfile来包含其他的文件。

# Boost——STL的延伸

boost是一个准标准库，相当于STL的延续和扩充，它的设计理念和STL比较接近，都是利用泛型让复用达到最大化。不过对比STL，boost更加实用。STL集中在算法部分，而boost包含了不少工具类，可以完成比较具体的工作。

    boost主要包含一下几个大类：字符串及文本处理、容器、迭代子(Iterator)、[算法](http://lib.csdn.net/base/31)、函数对象和高阶编程、泛型编程、模板元编程、预处理元编程、并发编程、数学相关、纠错和测试、[数据结构](http://lib.csdn.net/base/31)、输入/输出、跨语言支持、内存相关、语法分析、杂项。 有一些库是跨类别包含的，就是既属于这个类别又属于那个类别。

### 文本处理

conversion/lexcial\_cast类用于“用C++”的方法实现数字类型和字符串之间的转换。 主要是替代C标准库中的 atoi、 itoa之类的函数。当然其中一个最大的好处就是支持泛型了。

### Format库

提供了对流的“printf-like”功能。printf里使用%d、%s等等的参数做替换的方法在很多情况下还是非常方便的，STL的iostream则缺乏这样的功能。format为stream增加了这个功能，并且功能比原始的printf更强。

### regex

这个不多说了，正则表达式库。如果需要做字符串分析的人就会理解正则表达式有多么有用了。

### Spirit

这个是做LL分析的框架，可以根据EBNF规则对文件进行分析。（不要告诉我不知道什么是EBNF）。做编译器的可能会用到。一般人不太用的到。

### tokenizer库

我以前经常在CSDN上看到有人问怎么把一个字符串按逗号分割成字符串数组。也许有些人很羡慕VB的split函数。现在，boost的tokenizer也有相同的功能了，如果我没记错的话，这个tokenizer还支持正则表达式，是不是很爽？

### array:

提供了常量大小的数组的一个包装，喜欢用数组但是苦恼数组定位、确定数组大小等功能的人这下开心了。

### dynamic\_bitset

动态分配大小的bitset，我们知道STL里有个bitset，为位运算提供了不少方便。可惜它的大小需要在编译期指定。现在好了，运行期动态分配大小的bitset来了。

### graph

提供了图的容器和相关算法。我还没有在程序中用到过图，需要用的人可以看看。

### multi\_array

提供了对多维数组的封装，应该还是比较有用的。

### 并发编程

只有一个库，thread，提供了一个可移植的线程库，不过在Windows平台上我感觉用处不大。因为它是基于Posix线程的，在Windows里对Posix的支持不是很好。

### 数学和数值

包含了很多数值处理方面的类库，数学类我也不太熟，不过这里有几个类还是很有用的，比如rational分数类，random随机数类，等等。

### static\_assert

提供了编译器的assert功能。

### test库

一个单元测试框架，非常不错。

### concept\_check

提供了泛型编程时，对泛型量的一点检查，不是很完善，不过比没有好。

### 数据类型类any

一个安全的可以包含不同对象的类。把它作为容器的元素类型，那么这个容器就可以包含不同类型的元素。比用void \*要安全。

### compressed\_pair

跟STL里的pair差不多。不过对空元素做了优化。

### Tuple

呵呵，也许是某些人梦寐以求的东西。可以让函数返回多个值。

### 跨语言支持：[Python](http://lib.csdn.net/base/11)

呵呵，好东东啊，可以将C++的类和函数映射给python使用。以下为几个CSDN上的关于boost.python的中文资料：http://dev.csdn.net/article/19/19828.shtm，http://dev.csdn.net/article/19/19829.shtm，http://dev.csdn.net/article/19/19830.shtm，http://dev.csdn.net/article/19/19831.shtm

### pool

内存池，呵呵，不用害怕频繁分配释放内存导致内存碎片，也不用自己辛辛苦苦自己实现了。

### smart\_ptr

智能指针，这下不用担心内存泄漏的问题了吧。不过，C++里的智能指针都还不是十全十美的，用的时候小心点了，不要做太技巧性的操作了。

### date\_time

这个是平台、类库无关的实现，如果程序需要跨平台，可以考虑用这个。

### timer

提供了一个计时器，虽然不是Windows里那种基于消息的计时器，不过据说可以用来测量语句执行时间。

### uitlity

提供了一个noncopyable类，可以实现“无法复制”的类。很多情况下，我们需要避免一个类被复制，比如代表文件句柄的类，文件句柄如果被两个实例共享，操作上会有很多问题，而且语义上也说不过去。一般的避免实例复制的方法是把拷贝构造和operator=私有化，现在只要继承一下这个类就可以了，清晰了很多。

### value\_initialized

数值初始化，可以保证声明的对象都被明确的初始化，不过这个真的实用吗？似乎写这个比直接写初始化还累。呵呵，仁者见仁了。

     这里面除了regex、python和test需要编译出库才能用，其他的大部分都可以直接源代码应用，比较方便。其实这些库使用都不难。最主要的原因是有些库的使用需要有相关的背景知识，比如元编程、STL、泛型编程等等。

# C++的HTTP服务器

cpp-netlib是一个使用现代C+技术提供应用层协议支持的库集合。它重量轻，速度快，便携，并打算尽可能容易配置。

# C++加密解密库

Crypto++库，开发者是Wei Dai，使用C++写的加密库，实现了非常多的加密算法，基本能满足我们的加密需求，使用起来也很简单方便，这是官方网站<http://www.cryptopp.com/>

# OpenSSL套接字数据加密

[SSL](https://baike.baidu.com/item/SSL)是Secure Sockets Layer（[安全套接层协议](https://baike.baidu.com/item/%E5%AE%89%E5%85%A8%E5%A5%97%E6%8E%A5%E5%B1%82%E5%8D%8F%E8%AE%AE)）的缩写，可以在Internet上提供秘密性传输。[Netscape](https://baike.baidu.com/item/Netscape)公司在推出第一个[Web浏览器](https://baike.baidu.com/item/Web%E6%B5%8F%E8%A7%88%E5%99%A8)的同时，提出了SSL协议标准。其目标是保证两个应用间通信的保密性和可靠性,可在服务器端和用户端同时实现支持。已经成为[Internet](https://baike.baidu.com/item/Internet)上保密通讯的工业标准。

### 核心功能

OpenSSL整个软件包大概可以分成三个主要的功能部分：SSL协议库、应用程序以及[密码算法](https://baike.baidu.com/item/%E5%AF%86%E7%A0%81%E7%AE%97%E6%B3%95)库。OpenSSL的目录结构自然也是围绕这三个功能部分进行规划的。

作为一个基于密码学的安全开发包，OpenSSL提供的功能相当强大和全面，囊括了主要的密码算法、常用的[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)和证书封装管理功能以及SSL协议，并提供了丰富的应用程序供测试或其它目的使用。

### 辅助功能

[BIO](https://baike.baidu.com/item/BIO)机制是OpenSSL提供的一种高层IO接口，该接口封装了几乎所有类型的[IO](https://baike.baidu.com/item/IO/5918)接口，如内存访问、文件访问以及Socket等。这使得代码的重用性大幅度提高，OpenSSL提供API的复杂性也降低了很多。

OpenSSL对于随机数的生成和管理也提供了一整套的解决方法和支持API函数。随机数的好坏是决定一个[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)是否安全的重要前提。

OpenSSL还提供了其它的一些辅助功能，如从口令生成密钥的API，证书签发和管理中的配置文件机制等等。如果你有足够的耐心，将会在深入使用OpenSSL的过程慢慢发现很多这样的小功能，让你不断有新的惊喜。

# C/S数据交互

### Protobuf是什么

Protobuf(Google Protocol Buffers)是google开发的的一套用于数据存储，网络通信时用于协议编解码的工具库.它和XML和Json数据差不多,把数据已某种形式保存起来.Protobuf相对与XML和Json的不同之处，它是一种二进制的数据格式，具有更高的传输，打包和解包效率.

### Protobuf的语法

#### proto文件中数据类型可以分为两大类:

复合数据类型 + 标准数据类型

复合数据类型包括：枚举和message类型

标准数据类型包含：整型，浮点，字符串等

#### 数据类型前面修饰词:

①required: 必须赋值，不能为空，否则该条message会被认为是“uninitialized”。build一个“uninitialized” message会抛出一个RuntimeException异常，解析一条“uninitialized” message会抛出一条IOException异常。除此之外，“required”字段跟“optional”字段并无差别。

②optional:字段可以赋值，也可以不赋值。假如没有赋值的话，会被赋上默认值。

③repeated: 该字段可以重复任意次数，包括0次。重复数据的顺序将会保存在protocol buffer中，将这个字段想象成一个可以自动设置size的数组就可以了。

#### 每个字段要给数字:

该Number是用来标记该字段在序列化后的二进制数据中所在的field，每个字段的Number在message内部都是独一无二的。也不能进行改变，否则数据就不能正确的解包。

#### 数据类型

数据类型这里可以去看Protobuf支持的基本数据类型的表

### 为什么使用protobuf

在一些场景下，数据需要在不同的平台，不同的程序中进行传输和使用，例如某个消息是用C++程序产生的，而另一个程序是用java写的，当前者产生一个消息数据时，需要在不同的语言编写的不同的程序中进行操作，如何将消息发送并在各个程序中使用呢？这就需要设计一种消息格式，常用的就有json和xmprotobuf出现的较晚。   
protobuf 的优点主要是简单，快。   
 protobuf将数据序列化为二进制之后，占用的空间相当小，基本仅保留了数据部分，下面章节马上提到。而xml和json会附带消息结构在数据中。   
 protobuf使用起来也方便，只需要反序列化就可以了，而不需要xml和json那样层层解析。

# C++压缩与解压缩

Snappy 是一个 C++ 的用来压缩和解压缩的开发包。其目标不是最大限度压缩或者兼容其他压缩格式，而是旨在提供高速压缩速度和合理的[压缩率](https://baike.baidu.com/item/%E5%8E%8B%E7%BC%A9%E7%8E%87/6435712)。Snappy 比 zlib 更快，但文件相对要大 20% 到 100%。在 64位模式的 Core i7 处理器上，可达每秒 250~500兆的压缩速度。