目录

命令行

**jar cvfm MyWordCount.jar manifest.data \*.class**

**汉字**

代码段

#include <stdio.h>

int main(int argc, char\* argv[])

{}

1. 概述

**enum Operations {**

**opReply = 1, /\* reply. responseTo is set. \*/**

**dbMsg = 1000, /\* generic msg command followed by a string \*/**

**dbUpdate = 2001, /\* update object \*/**

**dbInsert = 2002,**

**//dbGetByOID = 2003,**

**dbQuery = 2004,**

**dbGetMore = 2005,**

**dbDelete = 2006,**

**dbKillCursors = 2007**

**};**

gcc和g++异同点

**误区一:gcc只能编译c代码,g++只能编译c++代码**

两者都可以，但是请注意：

1.后缀为.c的，gcc把它当作是C程序，而g++当作是c++程序；后缀为.cpp的，两者都会认为是c++程序，注意，虽然c++是c的超集，但是两者对语法的要求是有区别的。C++的语法规则更加严谨一些。

2.编译阶段，g++会调用gcc，对于c++代码，两者是等价的，但是因为gcc命令不能自动和C＋＋程序使用的库联接，所以通常用g++来完成链接，为了统一起见，干脆编译/链接统统用g++了，这就给人一种错觉，好像cpp程序只能用g++似的。

后缀为.c的，gcc把它当作是C程序，而g++当作是c++程序；后缀为.cpp的，两者都会认为是c++程序，注意，虽然c++是c的超集，但是两者对语法的要求是有区别的，例如：

[cpp] [view plain](http://blog.csdn.net/hansel/article/details/5008692) [copy](http://blog.csdn.net/hansel/article/details/5008692)

#include <stdio.h>

int main(int argc, char\* argv[]) {

   if(argv == 0) return;

   printString(argv);

   return;

}

int printString(char\* string) {

  sprintf(string, "This is a test./n");

}

如果按照C的语法规则，OK，没问题，但是，一旦把后缀改为cpp，立刻报三个错：“printString未定义”；

“cannot convert `char\*\*' to `char\*”；

”return-statement with no value“；

分别对应前面红色标注的部分。可见C++的语法规则更加严谨一些。

2.编译阶段，g++会调用gcc，对于c++代码，两者是等价的，但是因为gcc命令不能自动和C＋＋程序使用的库联接，所以通常用g++来完成链接，为了统一起见，干脆编译/链接统统用g++了，这就给人一种错觉，好像cpp程序只能用g++似的。

**误区二:gcc不会定义\_\_cplusplus宏，而g++会**

实际上，这个宏只是标志着编译器将会把代码按C还是C++语法来解释，如上所述，如果后缀为.c，并且采用gcc编译器，则该宏就是未定义的，否则，就是已定义。

**误区三:编译只能用gcc，链接只能用g++**

严格来说，这句话不算错误，但是它混淆了概念，应该这样说：编译可以用gcc/g++，而链接可以用g++或者gcc -lstdc++。因为gcc命令不能自动和C＋＋程序使用的库联接，所以通常使用g++来完成联接。但在编译阶段，g++会自动调用gcc，二者等价。

**误区四:extern "C"与gcc/g++有关系**

实际上并无关系，无论是gcc还是g++，用extern "c"时，都是以C的命名方式来为symbol命名，否则，都以c++方式命名。试验如下：

[cpp] [view plain](http://blog.csdn.net/hansel/article/details/5008692) [copy](http://blog.csdn.net/hansel/article/details/5008692)

**me.h**：

extern "C" void CppPrintf(void);

me.cpp:

#include <iostream>

#include "me.h"

using namespace std;

void CppPrintf(void)

{

cout << "Hello/n";

}

**test.cpp**:

#include <stdlib.h>

#include <stdio.h>

#include "me.h"

int main(void)

{

    CppPrintf();

    return 0;

}

1. 先给me.h加上extern "C"，看用gcc和g++命名有什么不同

[root@root G++]# g++ -S me.cpp

[root@root G++]# less me.s

.globl \_Z9CppPrintfv        //注意此函数的命名

        .type   CppPrintf, @function

[root@root GCC]# gcc -S me.cpp

[root@root GCC]# less me.s

.globl \_Z9CppPrintfv        //注意此函数的命名

        .type   CppPrintf, @function

完全相同！

2. 去掉me.h中extern "C"，看用gcc和g++命名有什么不同

[root@root GCC]# gcc -S me.cpp

[root@root GCC]# less me.s

.globl \_Z9CppPrintfv        //注意此函数的命名

        .type   \_Z9CppPrintfv, @function

[root@root G++]# g++ -S me.cpp

[root@root G++]# less me.s

.globl \_Z9CppPrintfv        //注意此函数的命名

        .type   \_Z9CppPrintfv, @function

完全相同！

调试MongoDB程序下断点出现错误：

出现错误

No source file named math/add.cpp.

Make breakpoint pending on future shared library load? (y or [n]) n

原因有俩：（1）编译连接是未加-g选项生成相关文件（2）gdb的源码搜索路径不对。

（1）加上-g选项编译链接会生成符号文件，这样在gdb调试的时候，才能查看源文件，在源文件中下断点，否则将无法下断点。

g++ -g \*.cpp

（2）-d是添加源文件路径

gdb -d src\_path ./mongod

修改后berak math/add.cpp:5应该就ok了。

-g系统原生格式的调试信息被写入二进制，其他调试器也可以使用这些调试信息。

-ggdb转为gcc/g++生成的调试信息，该调试信息不能用其他调试工具调试。

-g2/3/1，2是默认级别，产生符号表、行号、局部外部变量信息。3包含2中的信息及宏。1产生信息较少。

ps aux | grep mongod | grep -v gdb | grep -v grep | awk '{print $2}'

gdb调试程序

**调试带参数程序**

set args --dbpath /home/john/MongoDB2.6.12/mongodb-src-r2.6.12/db\_path/

mongoDbMain函数中

break src/mongo/db/db.cpp:1024

assembleResponse函数中

break src/mongo/db/instance.cpp:348

break src/mongo/db/db.cpp:763

find ./src/mongo/db | xargs grep -n "execOneInsert" | grep -v uassert

插入记录调试设置的断点

break src/mongo/db/instance.cpp:1024 DBDirectClient::call(…)->assembleResponse( toSend, dbResponse , \_clientHost );

break src/mongo/db/instance.cpp:1037 DBDirectClient::say(…)assembleResponse( toSend, dbResponse , \_clientHost );

break src/mongo/db/db.cpp:205 MyMessageHandler::process(…)->assembleResponse( m, dbresponse, port->remote() );

break src/mongo/db/instance.cpp:445 receivedQuery(c , dbresponse, m );

break src/mongo/db/instance.cpp:275 dbresponse.exhaustNS = newRunQuery(m, q, op, \*resp); 在此未停

src/mongo/db/query/new\_find.cpp:433 runCommands(ns, q.query, curop, bb, cmdResBuf, false, q.queryOptions)

break src/mongo/db/query/new\_find.cpp:125 return \_runCommands(ns, jsobj, b, anObjBuilder, fromRepl, queryOptions);

break src/mongo/db/repl/oplog.cpp:679 \_runCommands(ns, o, bb, ob, true, 0);

break src/mongo/db/dbcommands.cpp:1725 Command::execCommand(c, client, queryOptions, ns, jsobj, anObjBuilder, fromRepl);

break src/mongo/db/dbcommands.cpp:1595 retval = \_execCommand(c, dbname, cmdObj, queryOptions, errmsg, result, fromRepl);

break src/mongo/db/dbcommands.cpp:1620 retval = \_execCommand(c, dbname, cmdObj, queryOptions, errmsg, result, fromRepl);

break src/mongo/db/dbcommands.cpp:1652 retval = \_execCommand(c, dbname, cmdObj, queryOptions, errmsg, result, fromRepl);

break src/mongo/db/dbcommands.cpp:1388 return c->run(dbname, cmdObj, queryOptions, errmsg, result, fromRepl);

break src/mongo/db/commands/write\_commands/write\_commands.cpp:146 writeBatchExecutor.executeBatch( request, &response );

break src/mongo/db/commands/write\_commands/batch\_executor.cpp:270 bulkExecute( request, &upserted, &writeErrors );

break src/mongo/db/commands/write\_commands/batch\_executor.cpp:741 execInserts( request, errors );

break src/mongo/db/commands/write\_commands/batch\_executor.cpp:875 execOneInsert(&state, &error);

break src/mongo/db/commands/write\_commands/batch\_executor.cpp:1103 insertOne(state, &result);

break src/mongo/db/commands/write\_commands/batch\_executor.cpp:1061 singleInsert(insertDoc, state->getCollection(), pregen, result);

break src/mongo/db/commands/write\_commands/batch\_executor.cpp:1141 StatusWith<DiskLoc> status = collection->insertDocument( docToInsert, true, pregen );

break src/mongo/db/catalog/collection.cpp:189 StatusWith<DiskLoc> status = \_insertDocument( docToInsert, enforceQuota, preGen, false );

break src/mongo/db/catalog/collection.cpp:231 StatusWith<DiskLoc> loc = \_recordStore->insertRecord( docToInsert.objdata(), docToInsert.objsize(), enforceQuota ? largestFileNumberInQuota() : 0 );

**插入断点的时候不要加入额外的./等之类的字符**，否则会出现找不到文件的情况。

线程统计

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 1 | 12905 | 主线程 |  |  |
| 2 | 12908 | 信号处理线程 |  |  |
| 3 | 12909 | 数据落盘线程 |  |  |
| 4 | 13154 | 文件预分配线程 |  |  |
| 5 | 13155 | 日志dur线程 |  |  |
| 6 | 13156 | js脚本解析引擎 |  |  |
| 7 | 13161 | 游标监视线程 | 报告内存使用情况，删除过时游标。 |  |
| 8 | 13162 | 周期性任务执行线程 | 据说是任务执行线程，其他线程向该线程提交任务。 |  |
| 9 | 13163 | TTL任务执行线程 | 删除过时索引。 |  |
| 10 | 13166 | getDeleter线程 |  |  |
| 11 | 13167 | 索引修复线程 | <Namespace, NamespaceDetails>这个结构是HashTable类型，很多的索引结构为了方便，都采用原生的map结构，map结构底层由红黑树实现，天然的索引结构。 | 运行完，立马结束。 |
| 12 | 13275 | 客户端线程 | assembleResponse  关注opwrite方法。 | 请求一开始回来到这里，第一次进入到这里执行完成就会把数据返回；  第2次进入，；  第2次调用到dbmsg.getns()时会切换到TTL线程，此时写入结果成功已经返回。  第3次调用，然后切换到TTL时，返回完成。 |
| 13 |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

显示参数show args

设置调试时相关文件路径

gdb <program>普通文件调试

gdb <program> core调试core文件

gdb <program> <pid>调试正运行的某个程序

-symbols <file>,-s <file>

从指定文件中读取符号表。

-se file

从指定文件中读取符号表信息，并把他用在可执行文件中。

-core <file>, -c <file>

调试时core dump的core文件。

-directory <directory>,-d <directory>

加入一个源文件的搜索路径。默认搜索路径是环境变量中PATH所定义的路径。

r运行

c继续

step进入函数内部

next不进入函数内部

下断点

break /home/john/MongoDB2.6.12/mongodb-src-r2.6.12/src/mongo/db/catalog/collection.cpp:200

break src/mongo/db/instance.cpp:891

break src/mongo/db/instance.cpp:func

break src/mongo/util/net/message\_server\_port.cpp:209

查看断点

info break，可以显示断点编号

删除断点

clear /home/john/MongoDB2.6.12/mongodb-src-r2.6.12/ src/mongo/db/instance.cpp:func

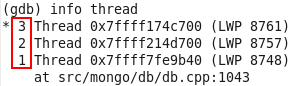
clear清除所有断点

delete break\_point\_num，根据断点编号删除断点

查看堆栈信息

bt

**info thread查看线程**



前面\*的是当前线程

thread 2切换到2号线程

break FileName.cpp:LinuNum thread all所有线程都在该文件中下断点

thread apply all command：所有线程都执行command命令

set scheduler-locking off|on|step：在调式某一个线程时，其他线程是否执行。off，不锁定任何线程，默认值。on，锁定其他线程，只有当前线程执行。step，在step（单步）时，只有被调试线程运行。

set non-stop on/off:启动不停模式，除了断点有关的线程会被停下来，其他线程都会正常执行，该选项GDB7.0之前不支持。

set pagination on/off:在使用backtrace时，在分页时是否停止。

set target-async on/ff:同步和异步。同步，gdb在输出提示符之前等待程序报告一些线程已经终止的信息。而异步的则是直接返回。

i locals  显示当前栈中所有变量

**多进程调试**

默认设置下，在调试多进程程序时GDB只会调试主进程。但是GDB（>V7.0）支持多进程的分别以及同时调试，换句话说，GDB可以同时调试多个程序。只需要设置follow-fork-mode(默认值：parent)和detach-on-fork（默认值：on）即可。

follow-fork-mode  detach-on-fork   说明

parent       on       只调试主进程（GDB默认）

child        on       只调试子进程

parent       off      同时调试两个进程，gdb跟主进程，子进程block在fork位置

child        off       同时调试两个进程，gdb跟子进程，主进程block在fork位置

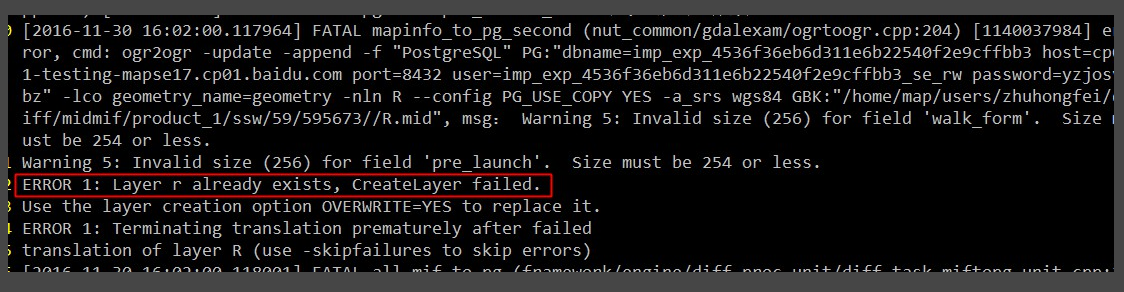
设置方法：set follow-fork-mode [parent|child]   set detach-on-fork [on|off]

查询正在调试的进程：info inferiors

切换调试的进程： inferior <infer number>

添加新的调试进程： add-inferior [-copies n] [-exec executable] ,可以用file executable来分配给inferior可执行文件。

其他：remove-inferiors infno， detach inferior



1. 参考资料

<http://www.cnblogs.com/qq78292959/archive/2012/12/29/2838882.html>，按照该链接的方式就可以编译成功。我安装时中间有几个报错，但不会影响MongoDB源码的编译。

scons mongod只编译mongod

scons all编译所有工具

http://blog.csdn.net/kangroger/article/details/47986197

http://blog.csdn.net/pbymw8iwm/article/details/7876797

http://blog.csdn.net/hansel/article/details/5008692

http://blog.csdn.net/haoel/article/details/2879

