北京大学 PEKING UNIVERSITY

瀚高 HIGH GO

透明数据库加密和密钥管理

开源开发实践-第二周

David & Cary

CONTENT
目 录

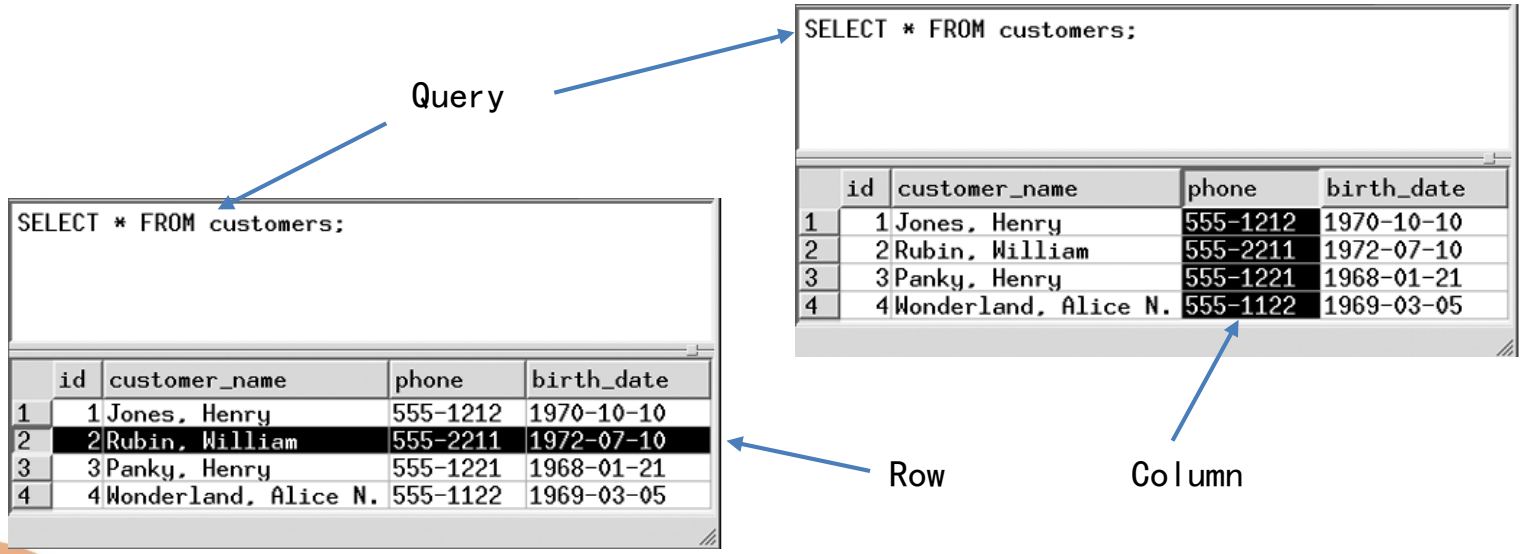
- 数据库常用术语
- 体系架构
- 源码编译
- 使用PostgreSQL
- 内核主要模块
- GDB调试工具
- 概括

数据库常用术语

数据库常用术语

了解数据库产业常用的术语

- 一个数据库牵扯到的技术很广泛，所以也牵扯到了很多在这行业常用到的专业术语
- 先了解这些基本专业术语是非常重要的



数据库常用术语

了解数据库产业常用的术语

Terminology	Definition
ACID	是指数据库管理系统（DBMS）在写入或更新资料的过程中，为保证事务（transaction）是正确可靠的，所必须具备的四个特性：原子性（atomicity，或称不可分割性）、一致性（consistency）、隔离性（isolation，又称独立性）、持久性（durability）
Transaction 事务	A transaction is a collection of database operations that are treated as a unit. PostgreSQL guarantees that all the operations within a transaction complete or that none of them complete
Database数据库	A database is a collection of tables. A database also contain views, indexes, sequences, data types, operators, and functions
Table 表	• A table is a collection of rows, and it has a name • A tables can be created as temporary and exist only to carry out a command. • All the rows in a table have the same shape (same set of columns). • Also known as a relation, relation file, or class
Row 行	• A row is a collection of column values. • Every row in a table has the same shape (same set of columns) • Also known as a tuple, or record

数据库常用术语

了解数据库产业常用的术语

Terminology	Definition
Column 列	• A column is the smallest unit of storage in a relational database. • A column represents one piece of information about an object. • Every column has a name and a data type • Also known as a field or attribute
Query 语句	A query is a type of command that retrieves data from the server.
Command 命令	A command is a string that you send to the server to get the server to do something useful. Some people use the word statement to mean command. The two words are interchangeable.
Commit 提交	commit marks the successful end of a transaction. When you perform a commit, you are telling PostgreSQL that you have completed a unit of operation and that all the changes that you made to the database should become permanent.
Rollback 回滚	rollback marks the unsuccessful end of a transaction. When you roll back a transaction, you are telling PostgreSQL to discard any changes that you have made to the database (since the beginning of the transaction).

数据库常用术语

了解数据库产业常用的术语

Terminology	Definition
WAL 预写日志	• Write Ahead Logging is a standard method for ensuring data integrity • changes to data files must be written only after those changes have been logged and flushed to permanent storage
Shared Buffer 共享内存	• Space allocated in memory containing data blocks for faster read and write operation. • All PostgreSQL’s backend processes have access to shared buffers
REDO 前滚恢复	Roll-forward recovery is an act of recovering database data by applying WAL files in an event of database crash
Cluster 集群	Cluster is a collection of databases and it could mean everything inside the data folder created by ‘initdb’,which contains one or more databases inside.
Catalog Table 系统表	• Catalog tables are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information • Can be understood as: PostgreSQL’s database tables

体系架构

PostgreSQL 体系架构

系统架构（System Architecture）

- PostgreSQL 内部的体系架构是非常庞大的
- 每一个内部的模块都有很棒的设计元素并可以单独被拿出来研究一番
- 不管你的软件背景知识是什么，总有一个PostgreSQL内部的模块适合您去重点钻研
- 因为内部体系太过庞大，我们只把重点放在比较重要的那几块
- 这门课我们会把PostgreSQL安装在Ubuntu 18.04 版本的Linux操作系统上面。所以对基本Linux操作必须要有些了解

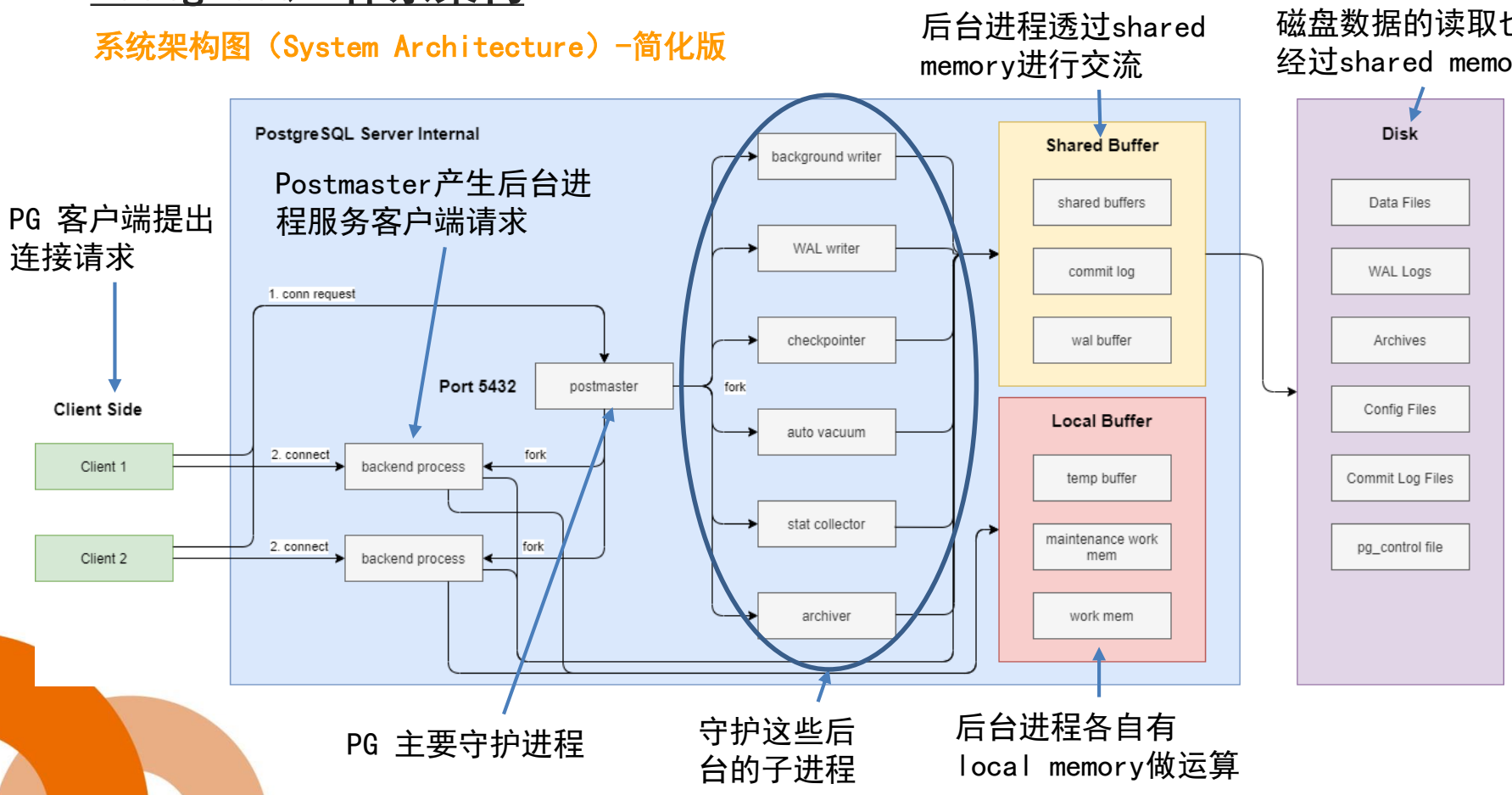
PostgreSQL 体系架构

Ubuntu Linux操作系统

- Ubuntu 18.04 LTS 可以透过这个链接下载.ISO 文件（https://releases.ubuntu.com/18.04/）
- 下载完了以后可以使用免费的vmware player或是virtualbox来读取。
  - 下载vmware：（https://www.vmware.com/ca/products/workstation-player/workstation-player-evaluation.html）
  - 下载virtualbox：（https://www.virtualbox.org/）
- 读取成功了以后就可以按照Ubuntu的安装程序一步一步把它安装完成
- 可以参考Ubuntu官方Linux指令指南（如果你有需要的话）：（https://ubuntu.com/tutorials/command-line-for-beginners#6-a-bit-of-plumbing）

PostgreSQL 体系架构

系统架构图（System Architecture）-简化版



PostgreSQL 体系架构

概括

- PostgreSQL服务器本身是由多个进程组合而成的
- 这些进程全部是被postmaster这个守护进程维护者，
- 上一页的架构图，只列了一些常看到了后台进程，还有许多其它的我们日后会再讨论到
- PG运用了传统的客户端和服务器架构来让用户访问数据库内容。
  - 客户端包含了PG自带的‘psql’客户端
  - 或是用户可以利用PG提供的‘libpq’代码库来编写自己的客户端
- PG和磁盘之间隔了一层shared memory（共享内存），大多数的数据都会被放在shared memory里面，因为访问速度块。PG后台的checkpointer进程就是负责定期的把数据从shared memory存到磁盘上。

源码编译

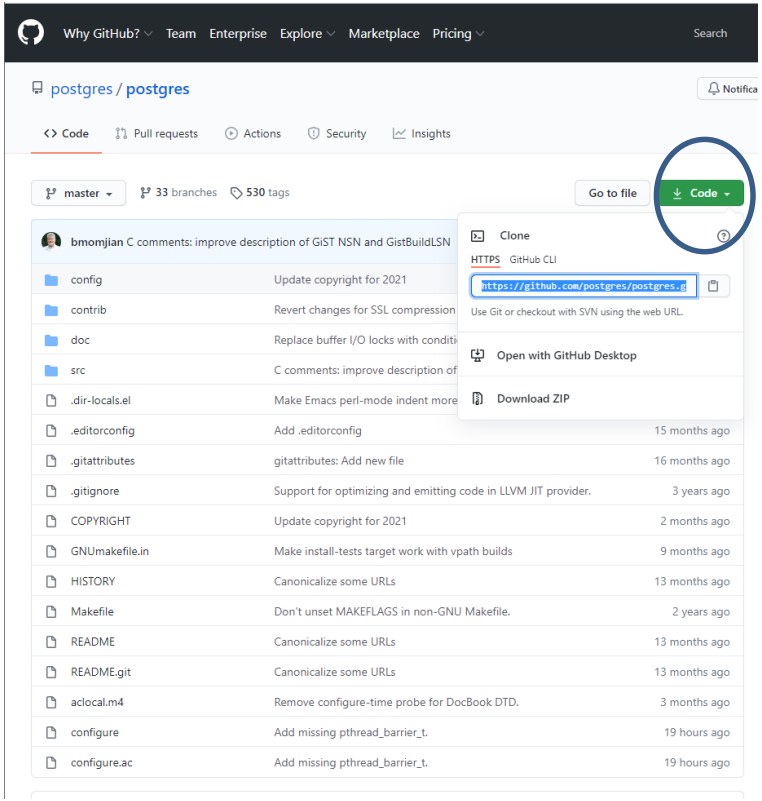
PostgreSQL 编译源码

下载官方PostgreSQL源代码

- 在开始使用PG之前，必须要先安装它
- 我们会再Ubuntu Linux环境下，下载PG源代码，然后编译
- 我们会使用源代码管理工具‘git’去做下载（clone）的动作
- 首先，先到PostgreSQL官方的github页面找到git URL：（https://github.com/postgres/postgres）
- 在Ubuntu Linux上，打开一个会话窗口，执行：

```
git clone https://github.com/postgres/postgres.git
```

- Git 应该会开始下载源代码



PostgreSQL 编译源码

下载官方PostgreSQL源代码

- 然后执行 git branch 查看当前的开发支路，应该是在master branch 上
- 切换到目前PG13稳定版本的开发支路上

```
git checkout REL_13_2
```

- 可以利用git tag 来查看所有发布的版本支路
- 在安装PG之前，还必须先安装一些PG依赖包：参考这里

（https://wiki.postgresql.org/wiki/Compile_and_Install_from_source_code）

- 在Ubuntu Linux上，可以透过这个指令安装依赖包

```
sudo apt-get install build-essential libreadline-dev zlib1g-dev flex bison libxml2-dev
libssl-dev libxml2-utils xsltproc
```

PostgreSQL 编译源码

下载官方PostgreSQL源代码

- （1）执行configure 脚本

```
./configure --prefix=$PWD/release --enable-debug
```

  - --prefix=给$PWD/release: 指定安装目录为当前目录下的release。
  - --enable-debug ： 保留语法符号，避免install时目标文件被strip掉了符号，调试时无法看到函数和变量名。
- 如果configure抱怨找不到依赖包，请参考上一页依赖包的安装
- 修改Makefile.global中的编译选项CFLAGS 把：-O2改为-O0，这样编译出来未优化，方便调试代码。
- （2）编译

```
make
```

- （3）安装

```
make install
```

PostgreSQL 编译源码

下载官方PostgreSQL源代码

- 安装完毕后，在你指定的安装目录下应该有几个新目录
  - bin
  - include
  - lib
  - share
- 我们接下来会利用到编译出来的二进制文件（在bin目录下的那些）来启动并连接PostgreSQL服务器

使用PostgreSQL

使用PostgreSQL

PostgreSQL Bin

- PostgreSQL官方版本提供了许多工具来操作，调试，查看当前PostgreSQL的工作状况
- 这些工具可以在你指定的安装目录下的bin目录里找到
- 在这里我们会讲一些比较常用的工具

```
caryh@HGPC01:~/highgo/git/projectv/sharedsm/highgo$ ls bin
clusterdb    dropuser  pg_archivecleanup  pg_config       pg_dumpall      pg_resetwal  pg_test_fsync  postgres    vacuumdb
createdb     ecpg      pg_basebackup      pg_controldata  pg_isready      pg_restore   pg_test_timing postmaster  vacuumlo
createuser   initdb    pgbench            pg_ctl          pg_receivewal   pg_rewind    pg_upgrade     psql
dropdb       oid2name  pg_checksums       pg_dump         pg_recvlogical  pg_standby   pg_waldump     reindexdb
```

使用PostgreSQL

初始化新的数据库

- 使用‘initdb’工具来初始化一个新的PostgreSQL数据库

```
Initdb  -D $name
```

- 可以随便指定一个名字来替代 $name
- 成功了以后，一个叫做$name的文件夹会在当前目录下被产生，这个文件夹就代表你的数据库集群
- 里面存放了很多PG相关的数据文件，例如配置文件等
- 可以用--help参数来看其他initdb的启动方法

```
Initdb --help
```

```
caryh@HGPC01:~/highgo/git/postgres.community2/postgres$ highgo/bin/initdb -D mydatabase
The files belonging to this database system will be owned by user "caryh".
This user must also own the server process.

The database cluster will be initialized with locale "en_CA.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

creating directory mydatabase ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... America/Vancouver
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

initdb: warning: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    highgo/bin/pg_ctl -D mydatabase -l logfile start

caryh@HGPC01:~/highgo/git/postgres.community2/postgres$
```

使用PostgreSQL

配置文件

- PostgreSQL主要的的配置文件在创建好的集群里面可以找到：

  `$name/postgresql.conf`

用来配置：

- PostgreSQL服务器的IP和端口号
- 共享内存大小
- WAL的生成即恢复的配置等等

参考资料

（https://www.postgresql.org/docs/current/runtime-config.html）

- 用户和链接权限可以透过另一个文件配置：

  `$name/pg_hba.conf`

用来配置：

- 哪一个用户可以访问那一个数据库？
- 访问数据库的IP白和黑名单
- 用户需要使用什么样的身份验证协议？
  - Password?
  - Radius?
  - TLS certificate?
  - GSSAPI?
  - LDAP? 等等

参考资料

（https://www.postgresql.org/docs/current/auth-pg-hba-conf.html）

使用PostgreSQL

启动PostgreSQL

- 当PostgreSQL集群被创建好了，配置文件也准备好了以后，就可以试着启动PostgreSQL服务了
- 可以利用‘pg_ctl’工具来启动或停止PG

```
pg_ctl  -D $name start
pg_ctl  -D $name stop
```

- $name 对应到的是你得集群文件夹的名字

- PG默认的服务端口号是‘5432’可以利用Linux的‘`netstat  -atun | grep 5432`’来查看是否有PG的服务在执行
- （注意）下面的例子用的是5431端口，你的应该是5432

```
caryh@HGPC01:~/highgo/git/postgres.community2/postgres$ netstat -atun | grep 5431
tcp        0      0 127.0.0.1:5431          0.0.0.0:*               LISTEN
caryh@HGPC01:~/highgo/git/postgres.community2/postgres$
```

```
caryh@HGPC01:~/highgo/git/postgres.community2/postgres$ highgo/bin/pg_ctl -D mydatabase/ start
waiting for server to start....2021-03-11 13:46:56.053 PST [989299] LOG:  starting PostgreSQL 12.5 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 7.4.0-1
ubuntu1~18.04.1) 7.4.0, 64-bit
2021-03-11 13:46:56.054 PST [989299] LOG:  listening on IPv4 address "127.0.0.1", port 5431
2021-03-11 13:46:56.060 PST [989299] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5431"
2021-03-11 13:46:56.081 PST [989300] LOG:  database system was shut down at 2021-03-11 12:34:24 PST
2021-03-11 13:46:56.086 PST [989299] LOG:  database system is ready to accept connections
 done
server started
```

使用PostgreSQL

查看启动后的PG后台程序

- 透过Ubuntu 的 ‘ps -ef | grep postgres’ 指令，我们可以看到postgres的守护进程和它的子进程

```
caryh    970808      1  0 15:28 ?        00:00:00 /home/caryh/highgo/git/postgres.community2/postgres/highgo/bin/postgres -D primary
caryh    970810 970808  0 15:28 ?        00:00:00 postgres: checkpointer
caryh    970811 970808  0 15:28 ?        00:00:00 postgres: background writer
caryh    970812 970808  0 15:28 ?        00:00:00 postgres: walwriter
caryh    970813 970808  0 15:28 ?        00:00:00 postgres: autovacuum launcher
caryh    970814 970808  0 15:28 ?        00:00:00 postgres: archiver
caryh    970815 970808  0 15:28 ?        00:00:00 postgres: stats collector
caryh    970816 970808  0 15:28 ?        00:00:00 postgres: logical replication launcher
```

这个是postmaster进程

每一个进程在Linux OS上都有一个process ID（简称pid）

这个是启动这个进程的进程，又称为parent pid。简单来说，是谁启动这个进程的

Parent pid = 1 = postmaster

这些是postmaster维护的进程

使用PostgreSQL

使用psql客户端访问PG服务器

- 当PostgreSQL集群被创建好了，并且启动了，我们可以利用PG自带的客户端‘psql’来访问数据库
- PG默认的数据库叫做‘postgres’
- 默认的用户名就是你当前的Linux OS的用户名

Example:

我的Linux用户名叫做caryh，我可以用这个指令来访问PG：

```
psql  -d postgres  -U caryh  -p 5431
```

- 联通了以后就可以直接输入SQL指令来操作数据库

```
postgres=# create table test (a int, b int);
CREATE TABLE
postgres=# create table test2 (a int, b char(20));
CREATE TABLE
postgres=# create table test3 (a int primary key, b int, c text);
CREATE TABLE
postgres=# \d
       List of relations
 Schema | Name  | Type  | Owner
--------+-------+-------+-------
 public | test  | table | caryh
 public | test2 | table | caryh
 public | test3 | table | caryh
(3 rows)

postgres=# insert into test values ( 55, 55);
INSERT 0 1
postgres=# select * from test;
 a  | b
----+----
 55 | 55
(1 row)

postgres=#
```

使用PostgreSQL

查看新启动的backend process来服务psql

- 再次透过Ubuntu 的 ‘ps -ef | grep postgres’ 指令，我们可以看到postmaster又多启动了一个backend process 来服务我们上一篇psql客户端的链接

```
caryh     989299       1  0 13:46 ?        00:00:00 /home/caryh/highgo/git/postgres.community2/postgres/highgo/bin/postgres -D mydatabase
caryh     989301  989299  0 13:46 ?        00:00:00 postgres: checkpointer
caryh     989302  989299  0 13:46 ?        00:00:00 postgres: background writer
caryh     989303  989299  0 13:46 ?        00:00:00 postgres: walwriter
caryh     989304  989299  0 13:46 ?        00:00:00 postgres: autovacuum launcher
caryh     989305  989299  0 13:46 ?        00:00:00 postgres: stats collector
caryh     989306  989299  0 13:46 ?        00:00:00 postgres: logical replication launcher
caryh     989605  988184  0 14:23 pts/1    00:00:00 highgo/bin/psql -d postgres -U caryh -p 5431
caryh     989606  989299  0 14:23 ?        00:00:00 postgres: caryh postgres [local] idle
caryh     989732  989721  0 14:26 pts/3    00:00:00 grep --color=auto postgres
caryh@HGPC01:~$
```

Postmaster新产生的 backend process

psql 客户端进程

使用PostgreSQL

其他常用的PG前端工具

Terminology	Definition
createdb	创建数据库
createuser	创建数据库用户
dropdb	删除数据库
dropuser	删除数据库用户
initdb	初始化PostgreSQL集群
psql	交互式PostgreSQL前端工具，可以用它来操作数据库

- 具体工具的使用方法可以参考PG官方文档说明

内核主要模块

使用PostgreSQL

语句处理主要流程

这门课的主要关注模块

返回结果

语法字面分析
- 检查字面上语法错误
- 生成语法树
- 入口在 parser/parser.c

语法深层分析
- 访问数据库
- 检查表是否存在
- 检查数据格式
- 把表名转换成内部OID
- 生成查询树
- 入口在 parser/analyze.c

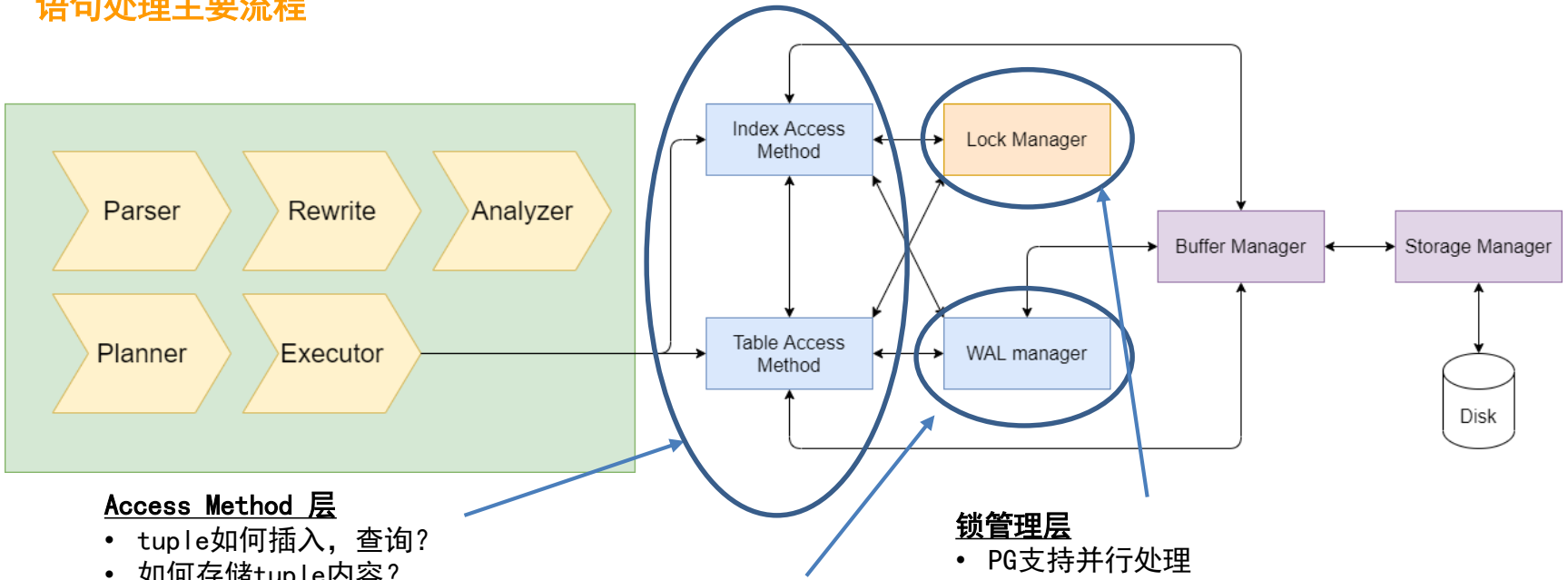
查询重写
通过查询语句的重写实现视图(view)和规则(rule)，所以需要的时候，在这个阶段会对查询语句进行重写
Rewrite/rewriteHandler.c

生成执行计划
- 选择可以在最短时间内完成查询的方法
- 入口在 optimizer/plan/planner.c

执行查询
- 执行执行计划里的步骤
- 入口在 executor/execMain.c

使用PostgreSQL

语句处理主要流程



Access Method 层

- tuple如何插入，查询？
- 如何存储tuple内容？
- 索引如何插入，查询？
- 如何创建新表？
- 如何删除，更新数据和索引？

WAL管理层

- 所有对数据库的操作都会透过WAL manager写到磁盘上

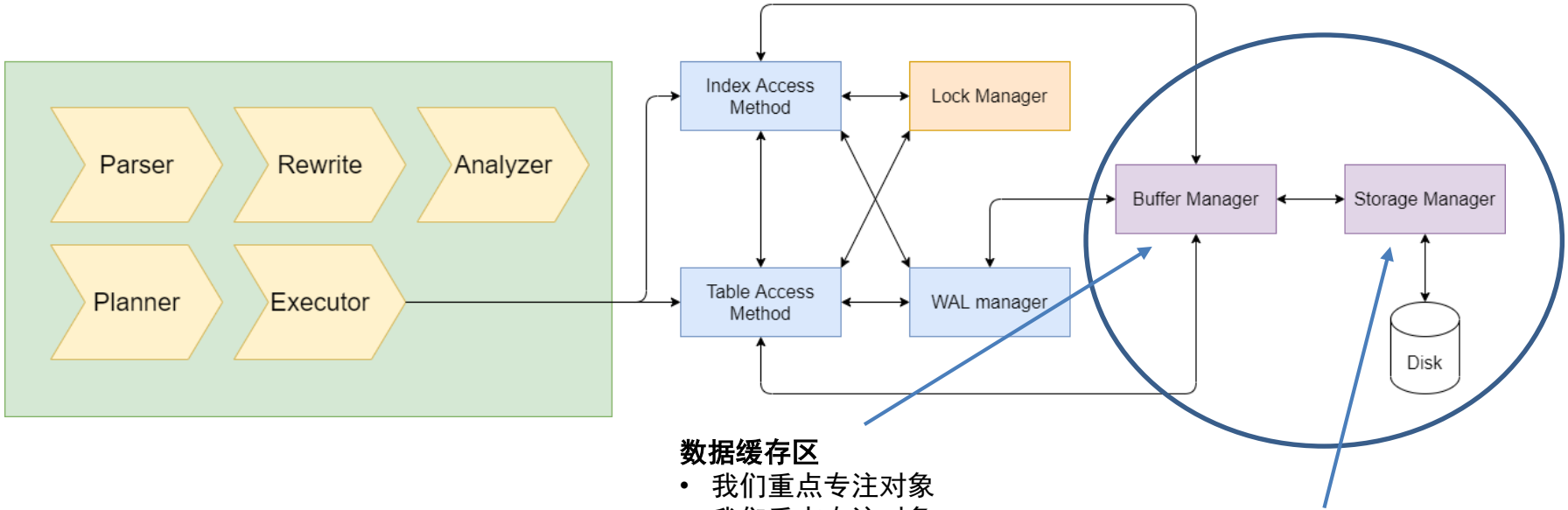
锁管理层

- PG支持并行处理
- 操作数据快之前都必须上锁
- 支持多种数据锁：
  - IOLock
  - ContentLock
  - BufferLock

使用PostgreSQL

语句处理主要流程

透明数据加密影响到的主要模块！



数据缓存区

- 我们重点专注对象
- 我们重点专注对象
- 所有PG 后台的进程都会使用到Buffer Manager的服务
- PG使用了buffer快的概念，一块默认大小8k

存储管理员

- 我们重点专注对象
- 缓冲区里的数据快最终都会被推到磁盘上了
- 目前存储的数据都是明文的

北京大学 PEKING UNIVERSITY

瀚高 HIGH GO

GDB 调试工具

GDB 调试工具

什么是gdb?

- GDB 全称“GNU symbolic debugger”
- 从名称上不难看出，它诞生于 GNU 计划（同时诞生的还有 GCC、Emacs 等），是 Linux 下常用的程序调试器。
- 发展至今，GDB 已经迭代了诸多个版本，实际场景中，GDB 更常用来调试 C 和 C++ 程序。
- Windows 操作系统中，人们更习惯使用一些已经集成好的开发环境（IDE），如 VS，VC，Dev-C++，eclipse 等，它们的内部已经嵌套了相应的调试器。

GDB 的吉祥物：弓箭鱼

GDB 调试工具

用GDB有什么好处？

- 最大的好处就是实时的断点调试
- 可在指定代码处暂停运行，并查看当前程序的运行状态（例如当前变量的值，函数的执行结果等）
- 可以改变某个变量的值，从而尝试修改程序中出现的逻辑错误。
- 除了调试和修复错误，另外一个好处就是帮助你了解程序运行的来龙去脉。
    - 从哪里开始，会经过哪里，到哪里结束。。。
    - 都可以看的一清二楚
    - 尤其是像PG这种庞大的系统，利用GDB先跟一轮可以大大的缩小看代码的范围

GDB 调试工具

用GDB有什么好处？

- 如果说。。。
  - Windows C/C++ 开发的一定要熟悉 Visual Studio
  - 从事 Java 开发的要熟悉 Eclipse 或 IntelliJ IDEA
  - 从事 Android 开发的要熟悉 Android Studio
  - 从事 iOS 开发的要熟悉 Xcode
- 那么从事 Linux C/C++ 开发的…那就要熟悉 GDB。

GDB 调试工具

GDB 常用命令

Terminology	Short form	Definition
list	l	查看源码
backtrace	bt	打印函数调用栈 （function callstack）
next	n	执行下一行
step	s	进入到函数，或执行下一行
fly	f	跳到指定callstack
continue	c	继续执行
breakpoint	b	设置断点
info breakpoints	Info b	显示断点信息
delete	d	删除断点
print	p	打印变量值
examine	x	打印内存值
run	r	启动程序
Ctrl-c		暂停程序，并调出GDB会话窗口

GDB 调试工具

使用GDB

- 必须先编译即安装好PostgreSQL。切记一定要在configure是设定 ‘enable_debug’ 参数还有取消compiler optimization （-O0）
- 安装GDB:Ubuntu Linux 本身就自带了GDB工具
- 找到想调试的 PostgreSQL backend 的进程号，我们选择服务psql客户端的backend process 为例子 （pid = 989606）

```
caryh     989299       1  0 13:46 ?        00:00:00 /home/caryh/highgo/git/postgres.community2/postgres/highgo/bin/postgres -D mydatabase
caryh     989301  989299  0 13:46 ?        00:00:00 postgres: checkpointer
caryh     989302  989299  0 13:46 ?        00:00:00 postgres: background writer
caryh     989303  989299  0 13:46 ?        00:00:00 postgres: walwriter
caryh     989304  989299  0 13:46 ?        00:00:00 postgres: autovacuum launcher
caryh     989305  989299  0 13:46 ?        00:00:00 postgres: stats collector
caryh     989306  989299  0 13:46 ?        00:00:00 postgres: logical replication launcher
caryh     989605  988184  0 14:23 pts/1    00:00:00 highgo/bin/psql -d postgres -U caryh -p 5431
caryh     989606  989299  0 14:23 ?        00:00:00 postgres: caryh postgres [local] idle
caryh     989732  989721  0 14:26 pts/3    00:00:00 grep --color=auto postgres
caryh@HGPC01:~$
```

GDB 调试工具

使用GDB

- 启动gdb，并告诉它想要调试的程序二进制文件位置。For example:

```
gdb $PWD/release/bin/postgres
```

- 在gdb的会话窗口里，附着到目标PID

```
（gdb）attach 989606
```

- 在gdb的会话窗口里，设置一个断点。For example:

```
（gdb）b heap_insert （函数名）
```

或

```
（gdb）b heapam.c:1840 （文件名加行数）
```

- 设定完毕后让程序继续进行

```
(gdb) c
```

GDB 调试工具

使用GDB

- 在另一个跑‘psql’客户端的窗口上，执行一个插入语具来触发gdb的breakpoint

```
INSERT INTO test VALUES （100,100);
```

- 此时，gdb管理的程序应该会被暂停，然后停在指定的断点上
- 客户端还在等待插入结果的返还。
- 这个时候我们就可以用gdb查看这个语句处理的来龙去脉。。。

GDB 调试工具

使用GDB

使用backtrace （bt）命令去查程序从哪里走到了我们的断点

我们设定的断点（breakpoint）

中间经过的所有函数调用每一个callstack都有编号，文件名，及行数

逻辑触发的起点

```
(gdb) bt
#0  heap_insert (relation=relation@entry=0x7f872f532228, tup=tup@entry=0x55ba8290f778, cid=0, options=options@entry=0,
    bistate=bistate@entry=0x0) at heapam.c:1840
#1  0x000055ba81ccde3e in simple_heap_insert (relation=relation@entry=0x7f872f532228, tup=tup@entry=0x55ba8290f778)
    at heapam.c:2356
#2  0x000055ba81d7826d in CatalogTupleInsert (heapRel=0x7f872f532228, tup=0x55ba8290f778) at indexing.c:228
#3  0x000055ba81d946ea in TypeCreate (newTypeOid=newTypeOid@entry=0, typeName=typeName@entry=0x7ffcf56ef820 "test",
    typeNamespace=typeNamespace@entry=2200, relationOid=relationOid@entry=16392, relationKind=relationKind@entry=114 'r',
    ownerId=ownerId@entry=16385, internalSize=-1, typeType=99 'c', typeCategory=67 'C', typePreferred=false,
    typDelim=44 ',', inputProcedure=2290, outputProcedure=2291, receiveProcedure=2402, sendProcedure=2403,
    typmodinProcedure=0, typmodoutProcedure=0, analyzeProcedure=0, elementType=0, isImplicitArray=false, arrayType=16393,
    baseType=0, defaultTypeValue=0x0, defaultTypeBin=0x0, passedByValue=false, alignment=100 'd', storage=120 'x',
    typeMod=-1, typNDims=0, typeNotNull=false, typeCollation=0) at pg_type.c:484
#4  0x000055ba81d710bc in AddNewRelationType (new_array_type=16393, new_row_type=<optimized out>, ownerid=<optimized out>,
    new_rel_kind=<optimized out>, new_rel_oid=<optimized out>, typeNamespace=2200, typeName=0x7ffcf56ef820 "test")
    at heap.c:1033
#5  heap_create_with_catalog (relname=relname@entry=0x7ffcf56ef820 "test", relnamespace=relnamespace@entry=2200,
    reltablespace=reltablespace@entry=0, relid=16392, relid@entry=0, reltypeid=reltypeid@entry=0,
    reloftypeid=reloftypeid@entry=0, ownerid=16385, accessmtd=2, tupdesc=0x55ba8287c620, cooked_constraints=0x0,
    relkind=114 'r', relpersistence=112 'p', shared_relation=false, mapped_relation=false, oncommit=ONCOMMIT_NOOP,
    reloptions=0, use_user_acl=true, allow_system_table_mods=false, is_internal=false, relrewrite=0, typaddress=0x0)
    at heap.c:1294
#6  0x000055ba81e3782a in DefineRelation (stmt=stmt@entry=0x55ba82876658, relkind=relkind@entry=114 'r', ownerId=16385,
    ownerId@entry=0, typaddress=typaddress@entry=0x0,
    queryString=queryString@entry=0x55ba82855648 "create table test (a int, b char(10)) using heap;") at tablecmds.c:885
#7  0x000055ba81fd5b2f in ProcessUtilitySlow (pstate=pstate@entry=0x55ba82876548, pstmt=pstmt@entry=0x55ba828565a0,
    queryString=queryString@entry=0x55ba82855648 "create table test (a int, b char(10)) using heap;",
    context=context@entry=PROCESS_UTILITY_TOPLEVEL, params=params@entry=0x0, queryEnv=queryEnv@entry=0x0, qc=0x7ffcf56efe50,
    dest=0x55ba82856860) at utility.c:1161
#8  0x000055ba81fd4120 in standard_ProcessUtility (pstmt=0x55ba828565a0,
    queryString=0x55ba82855648 "create table test (a int, b char(10)) using heap;", context=PROCESS_UTILITY_TOPLEVEL,
    params=0x0, queryEnv=0x0, dest=0x55ba82856860, qc=0x7ffcf56efe50) at utility.c:1069
#9  0x000055ba81fd1962 in PortalRunUtility (portal=0x55ba828b7dd8, pstmt=0x55ba828565a0, isTopLevel=<optimized out>,
    setHoldSnapshot=<optimized out>, dest=<optimized out>, qc=0x7ffcf56efe50) at pquery.c:1157
#10 0x000055ba81fd23e3 in PortalRunMulti (portal=portal@entry=0x55ba828b7dd8, isTopLevel=isTopLevel@entry=true,
    setHoldSnapshot=setHoldSnapshot@entry=false, dest=dest@entry=0x55ba82856860, altdest=altdest@entry=0x55ba82856860,
    qc=qc@entry=0x7ffcf56efe50) at pquery.c:1310
#11 0x000055ba81fd2f51 in PortalRun (portal=portal@entry=0x55ba828b7dd8, count=count@entry=9223372036854775807,
    isTopLevel=isTopLevel@entry=true, run_once=run_once@entry=true, dest=dest@entry=0x55ba82856860,
    altdest=altdest@entry=0x55ba82856860, qc=0x7ffcf56efe50) at pquery.c:779
#12 0x000055ba81fce967 in exec_simple_query (query_string=0x55ba82855648 "create table test (a int, b char(10)) using heap;")
    at postgres.c:1239
#13 0x000055ba81fd0d7e in PostgresMain (argc=<optimized out>, argv=argv@entry=0x55ba8287fdb0, dbname=<optimized out>,
    username=<optimized out>) at postgres.c:4315
#14 0x000055ba81f4f52a in BackendRun (port=0x55ba82877110, port=0x55ba82877110) at postmaster.c:4536
#15 BackendStartup (port=0x55ba82877110) at postmaster.c:4220
#16 ServerLoop () at postmaster.c:1739
#17 0x000055ba81f5063f in PostmasterMain (argc=3, argv=0x55ba8284fee0) at postmaster.c:1412
#18 0x000055ba81c91c04 in main (argc=3, argv=0x55ba8284fee0) at main.c:210
(gdb)
```

GDB 调试工具

使用GDB

使用fly 命令去跳到指定的callstack根据指定的编号

```
(gdb) f 3
#3  0x000055ba81d946ea in TypeCreate (newTypeOid=newTypeOid@entry=0, typeName=typeName@entry=0x7ffcf56ef820 "test",
    typeNamespace=typeNamespace@entry=2200, relationOid=relationOid@entry=16392, relationKind=relationKind@entry=114 'r',
    ownerId=ownerId@entry=16385, internalSize=-1, typeType=99 'c', typeCategory=67 'C', typePreferred=false,
    typDelim=44 ',', inputProcedure=2290, outputProcedure=2291, receiveProcedure=2402, sendProcedure=2403,
    typmodinProcedure=0, typmodoutProcedure=0, analyzeProcedure=0, elementType=0, isImplicitArray=false, arrayType=16393,
    baseType=0, defaultTypeValue=0x0, defaultTypeBin=0x0, passedByValue=false, alignment=100 'd', storage=120 'x',
    typeMod=-1, typNDims=0, typeNotNull=false, typeCollation=0) at pg_type.c:484
484                     CatalogTupleInsert(pg_type_desc, tup);
(gdb)
```

跳过去了以后，就可以使用print命令查看在callstack 3的时候函数及变量的状态

```
(gdb) p tup
$1 = (HeapTuple) 0x55ba8290f778
(gdb) p pg_type_desc
$2 = (Relation) 0x7f872f532228

(gdb)  p * tup
$3 = {t_len = 176, t_self = {ip_blkid = {bi_hi = 65535, bi_lo = 65535}, ip_posid = 0}, t_tableOid = 0,
  t_data = 0x55ba8290f790}

(gdb) p * pg_type_desc
$4 = {rd_node = {spcNode = 1663, dbNode = 16384, relNode = 1247}, rd_smgr = 0x55ba828e2a38, rd_refcnt = 2, rd_backend = -1,
  rd_islocaltemp = false, rd_isnailed = true, rd_isvalid = true, rd_indexvalid = true, rd_statvalid = false,
  rd_createSubid = 0, rd_newRelfilenodeSubid = 0, rd_firstRelfilenodeSubid = 0, rd_droppedSubid = 0,
  rd_rel = 0x7f872f532438, rd_att = 0x7f872f532548, rd_id = 1247, rd_lockInfo = {lockRelId = {relId = 1247, dbId = 16384}},
  rd_rules = 0x0, rd_rulescxt = 0x0, trigdesc = 0x0, rd_rsdesc = 0x0, rd_fkeylist = 0x0, rd_fkeyvalid = false,
  rd_partkey = 0x0, rd_partkeycxt = 0x0, rd_partdesc = 0x0, rd_pdcxt = 0x0, rd_partcheck = 0x0, rd_partcheckvalid = false,
  rd_partcheckcxt = 0x0, rd_indexlist = 0x7f872f477d00, rd_pkindex = 0, rd_replidindex = 0, rd_statlist = 0x0,
  rd_indexattr = 0x0, rd_keyattr = 0x0, rd_pkattr = 0x0, rd_idattr = 0x0, rd_pubactions = 0x0, rd_options = 0x0,
  rd_amhandler = 0, rd_tableam = 0x55ba82562c20 <heapam_methods>, rd_index = 0x0, rd_indextuple = 0x0, rd_indexcxt = 0x0,
  rd_indam = 0x0, rd_opfamily = 0x0, rd_opcintype = 0x0, rd_support = 0x0, rd_supportinfo = 0x0, rd_indoption = 0x0,
  rd_indexprs = 0x0, rd_indpred = 0x0, rd_exclops = 0x0, rd_exclprocs = 0x0, rd_exclstrats = 0x0, rd_indcollation = 0x0,
  rd_opcoptions = 0x0, rd_amcache = 0x0, rd_fdwroutine = 0x0, rd_toastoid = 0, pgstat_info = 0x55ba828d5cb0}
(gdb)
```

GDB 调试工具

使用GDB

使用print命令查到tuple的长度为176

使用examine命令检查tuple data在内存的数据

```
(gdb)  p *tup
$6 = {t_len = 176, t_self = {ip_blkid = {bi_hi = 65535, bi_lo = 65535}, ip_posid = 0}, t_tableOid = 0,
  t_data = 0x55ba8290f790}

(gdb)  p tup->t_data
$7 = (HeapTupleHeader) 0x55ba8290f790
(gdb) x/176bx  tup->t_data
0x55ba8290f790: 0xc0    0x02    0x00    0x00    0xff    0xff    0xff    0xff
0x55ba8290f798: 0x47    0x00    0x00    0x00    0xff    0xff    0xff    0xff
0x55ba8290f7a0: 0x00    0x00    0x1f    0x00    0x01    0x00    0x20    0xff
0x55ba8290f7a8: 0xff    0xff    0x0f    0x00    0x00    0x00    0x00    0x00
0x55ba8290f7b0: 0x0a    0x40    0x00    0x00    0x74    0x65    0x73    0x74
0x55ba8290f7b8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55ba8290f7c0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55ba8290f7c8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55ba8290f7d0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55ba8290f7d8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55ba8290f7e0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55ba8290f7e8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55ba8290f7f0: 0x00    0x00    0x00    0x00    0x98    0x08    0x00    0x00
0x55ba8290f7f8: 0x01    0x40    0x00    0x00    0xff    0xff    0x00    0x63
0x55ba8290f800: 0x43    0x00    0x01    0x2c    0x08    0x40    0x00    0x00
0x55ba8290f808: 0x00    0x00    0x00    0x00    0x09    0x40    0x00    0x00
0x55ba8290f810: 0xf2    0x08    0x00    0x00    0xf3    0x08    0x00    0x00
0x55ba8290f818: 0x62    0x09    0x00    0x00    0x63    0x09    0x00    0x00
0x55ba8290f820: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x55ba8290f828: 0x00    0x00    0x00    0x00    0x64    0x78    0x00    0x00
0x55ba8290f830: 0x00    0x00    0x00    0x00    0xff    0xff    0xff    0xff
0x55ba8290f838: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

GDB 调试工具

使用GDB

- 虽然GDB提供的调试命令有很多种，真正常用的就是这一些…
- 像是print，examine，fly，backtrace…
- 多加练习
- 你很快就可以上手的！

概括

概括

资源链接

- Ubuntu Linux 18.04

  (https://releases.ubuntu.com/18.04/)

- Vmware player

  (https://www.vmware.com/ca/products/workstation-player/workstation-player-evaluation.html)

- virtualbox

  (https://www.virtualbox.org/)

- Ubuntu官方Linux指令指南：

  (https://ubuntu.com/tutorials/command-line-for-beginners#6-a-bit-of-plumbing)

- PG github官方代码下载

  (https://github.com/postgres/postgres)

- PG 配置

  (https://www.postgresql.org/docs/current/runtime-config.html)

  (https://www.postgresql.org/docs/current/auth-pg-hba-conf.html)

概括

作业

- 搭建你自己的PG开发环境，包含:
  - 安装Ubuntu Linux 18.04
  - 使用git下载PG源代码，安装依赖包
  - 以自己喜好选择适合的代码编辑工具（IDE)，例如 eclipse for C
  - 在Linux上编译PostgreSQL，（debug 版本，无优化）安装并启动
  - 使用psql客户端做简单SQL操作 （完成后截个图）
  - 使用GDB调试PG，并成功看到代码的backtrace （完成后截个图）
- 在下次上课前把你的截图发给助教 Wendy 即可。

HIGH GO

融知与行 瀚且高远

THANKS