**Assignment One**

**Report**

-

09020328 Liang Wang

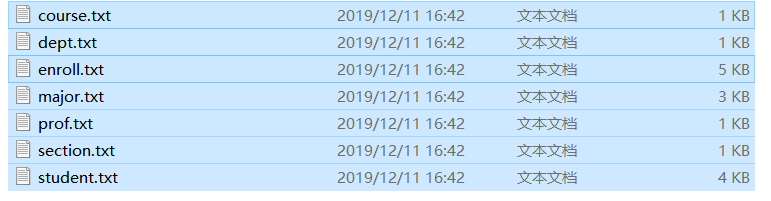
# **First: make the structure of the database**

This part resolves around some basic steps I took to set up the environment we need in the following parts.

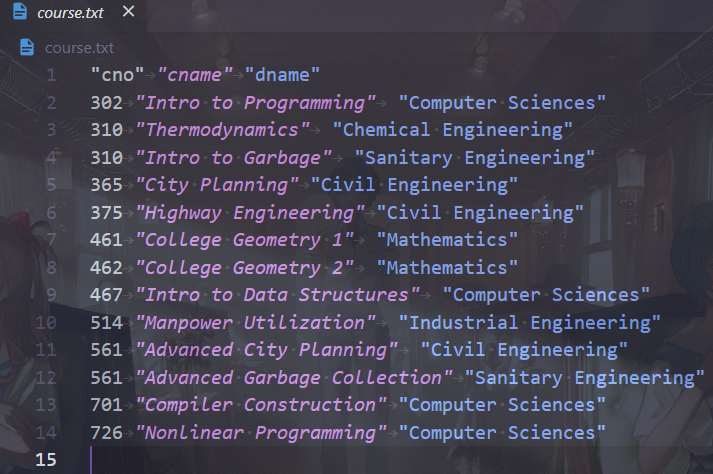
We start our journey by entering the entrance of Access and create our “university” database as requested. I stored this file to desktop purely out of convenience.



Before continuing to make progress in Access, Let’s switch back to check data files assigned which endows more than enough information to create all tables we need.

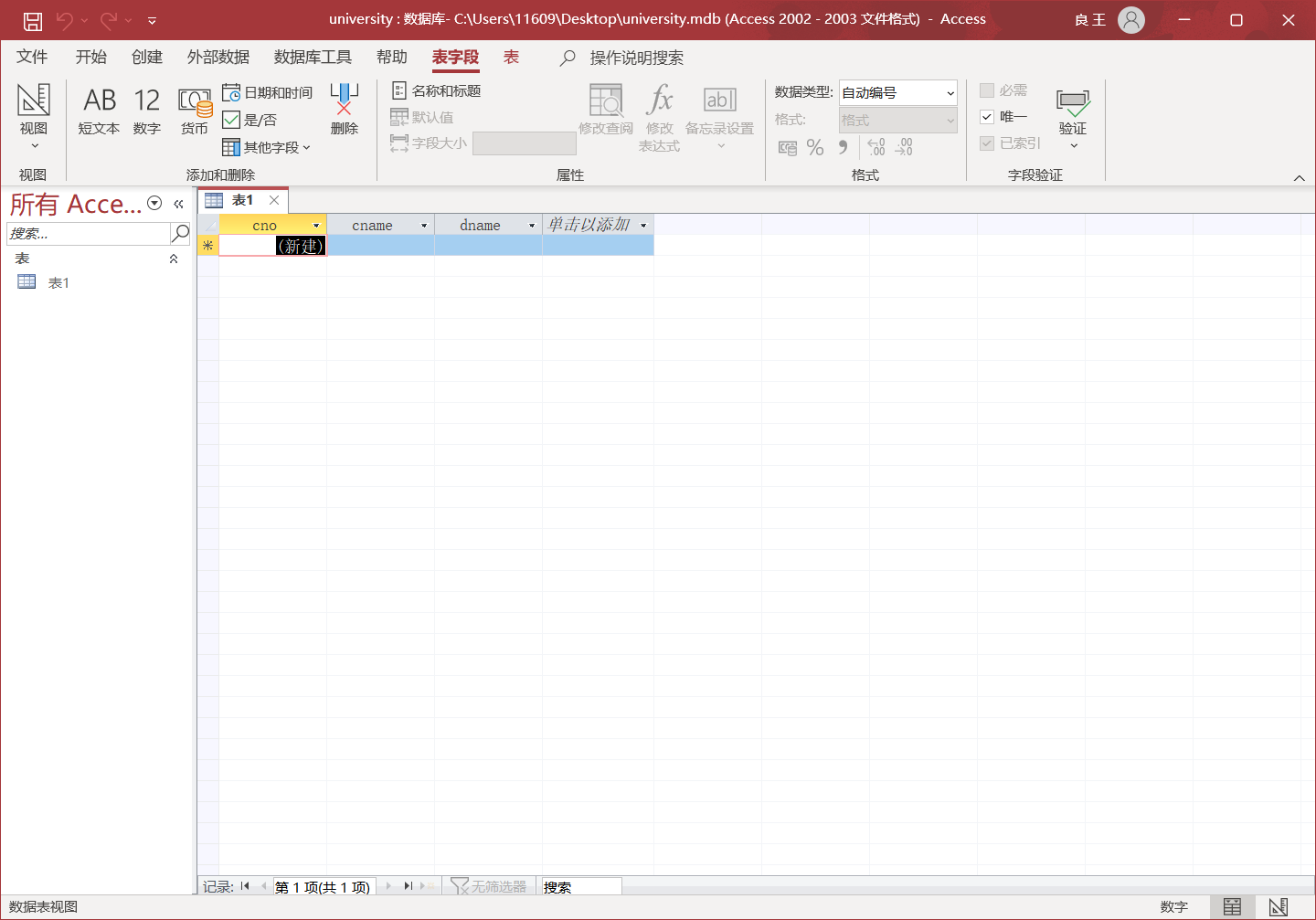


By analyzing the records of each file, we are confident to presume what type of each field is. But still out of convenience, I first assume all numbers like “302” to be number data type and all strings to be short-text. (course.txt to exemplify, applicable for others)

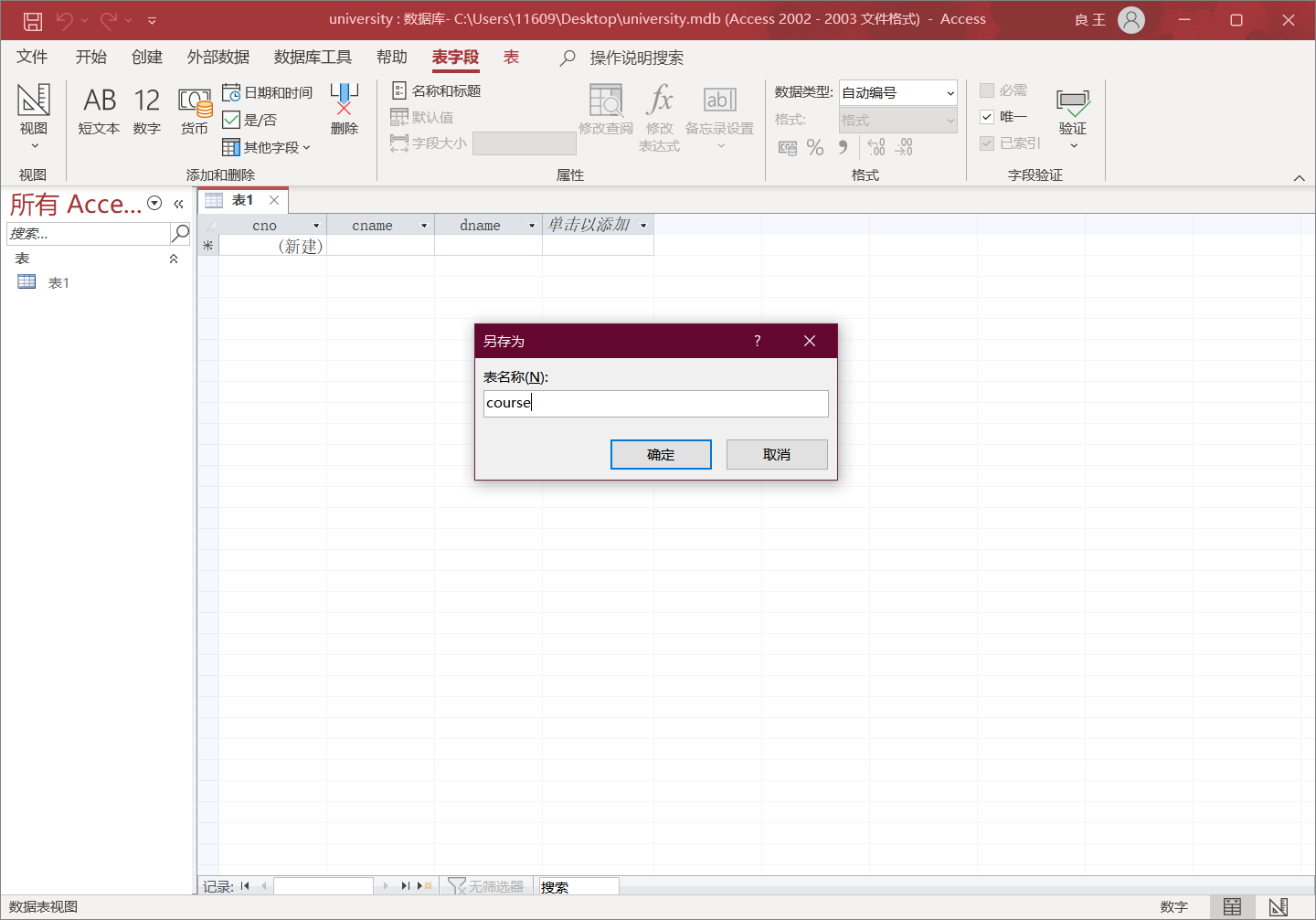


With the information provided, we can create the tables we need now. To accomplish that, we fill the table with the fields. (Here is still “course” table to exemplify)

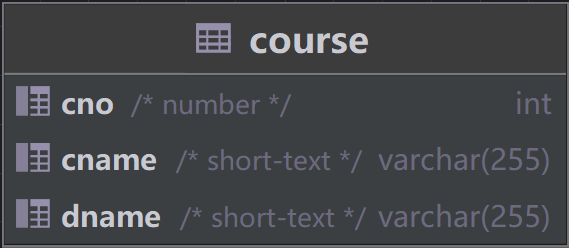
**(Note that auto-coding is not suitable here, we would dip into that later in this doc.)**

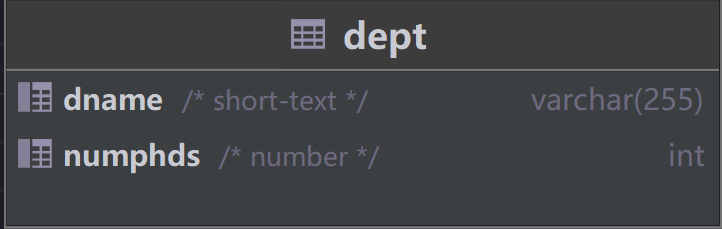


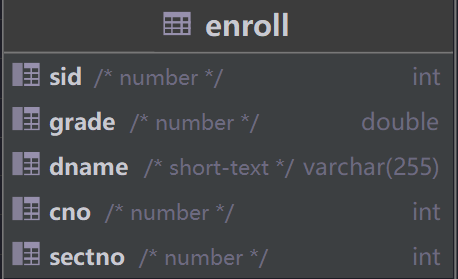
And save it by pressing ”Ctrl + s”.

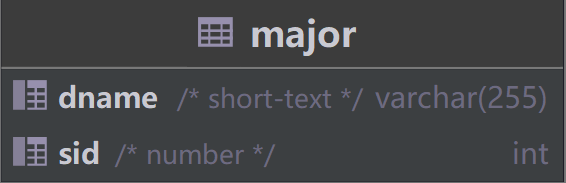


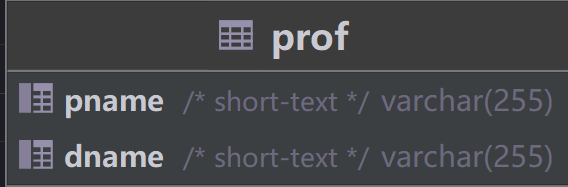
Repeat the steps for all files. And here is all we got, with all the schemas demonstrated down below ( a Mysql DDL sql file, which is the equivalent version of schemas of these tables in my perspective, is offered as well).

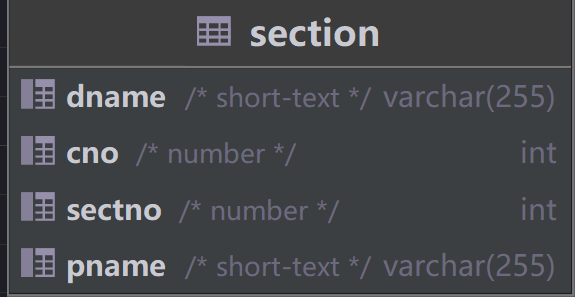


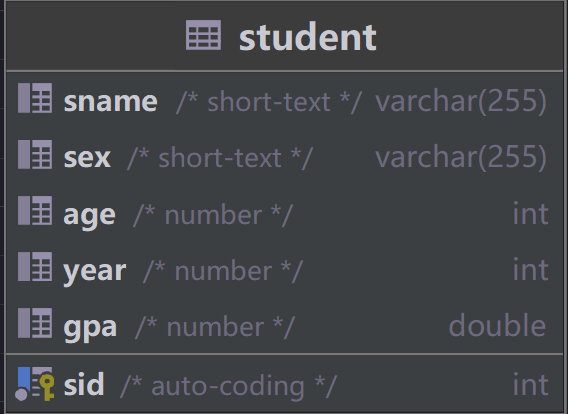




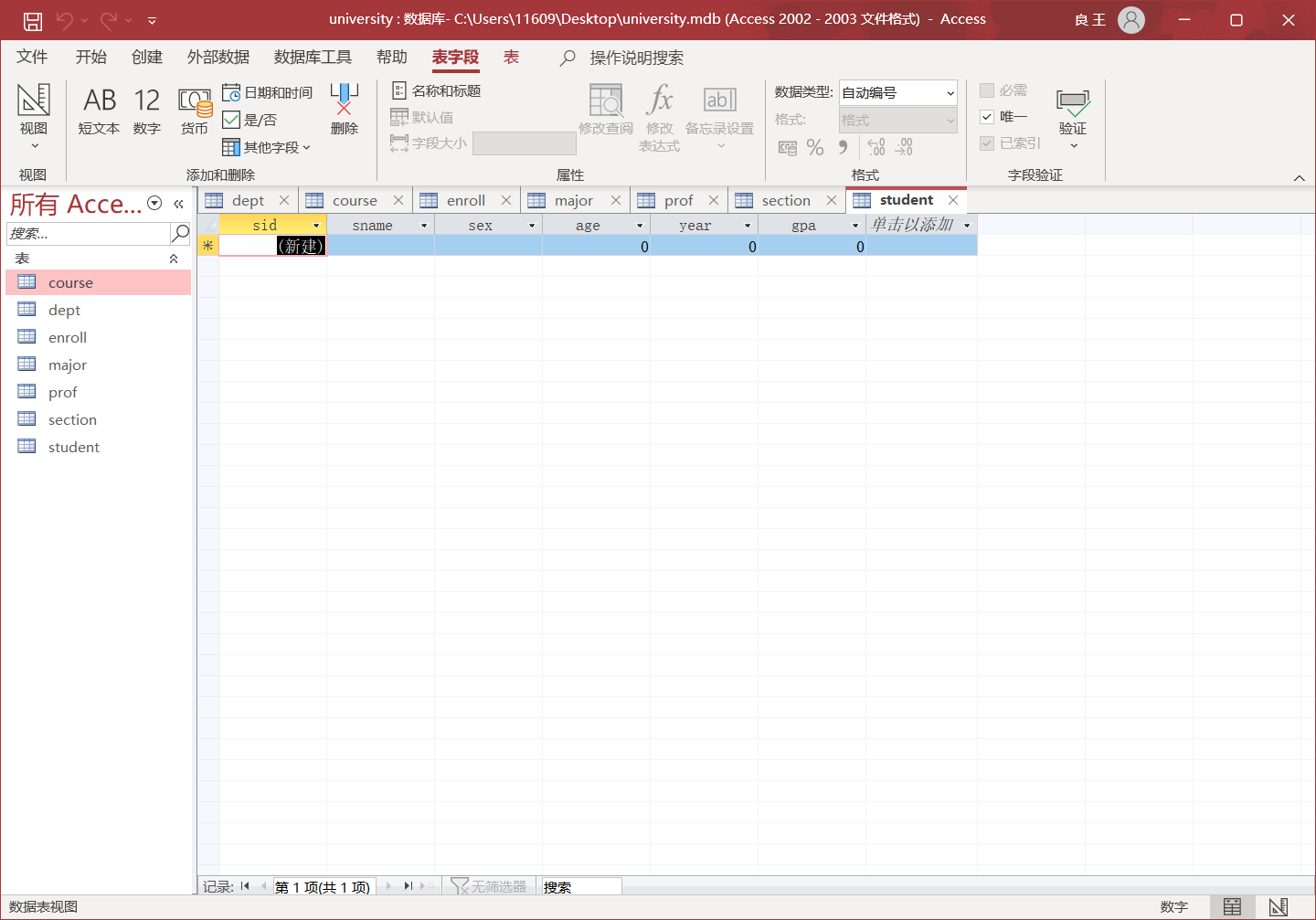






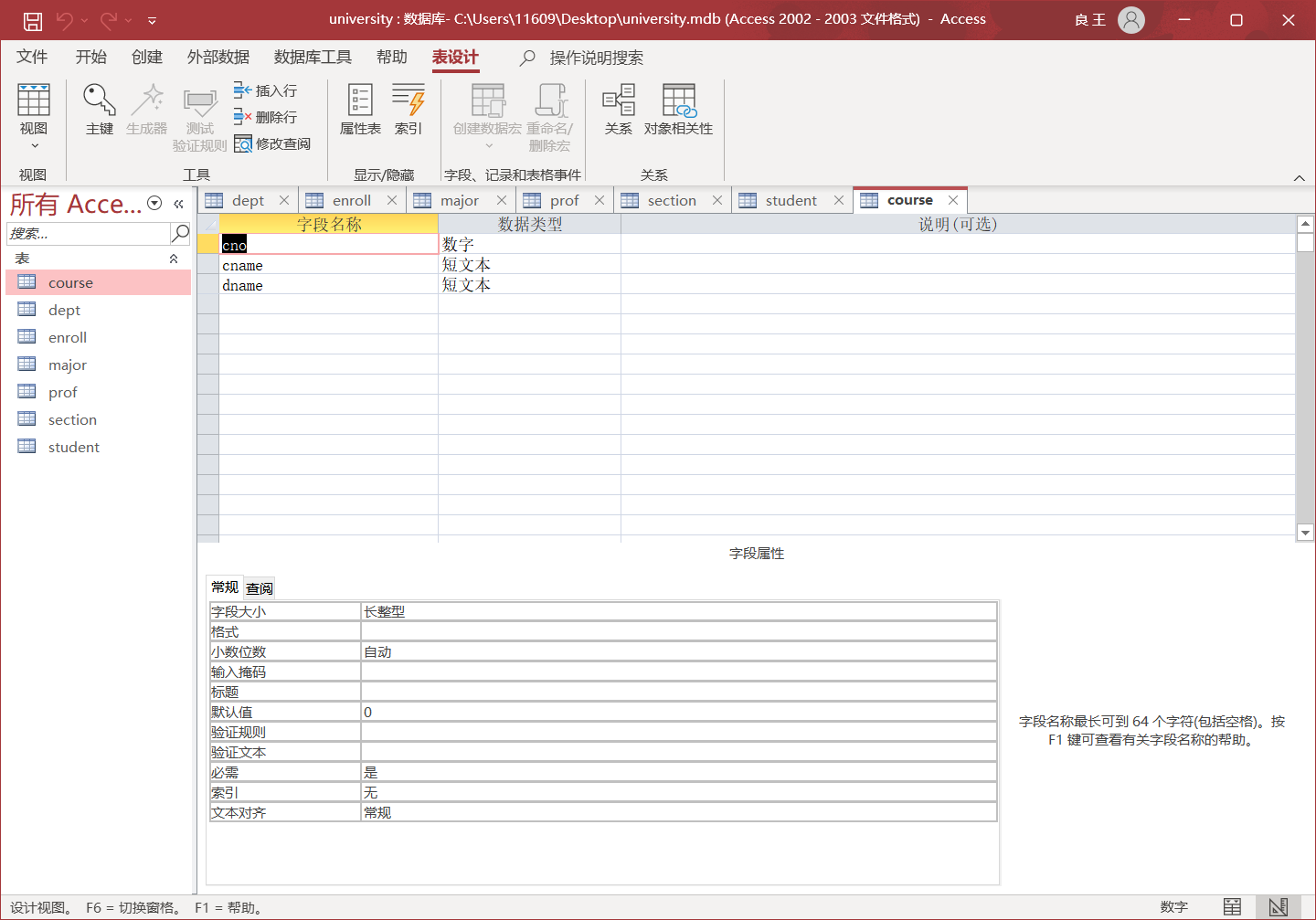


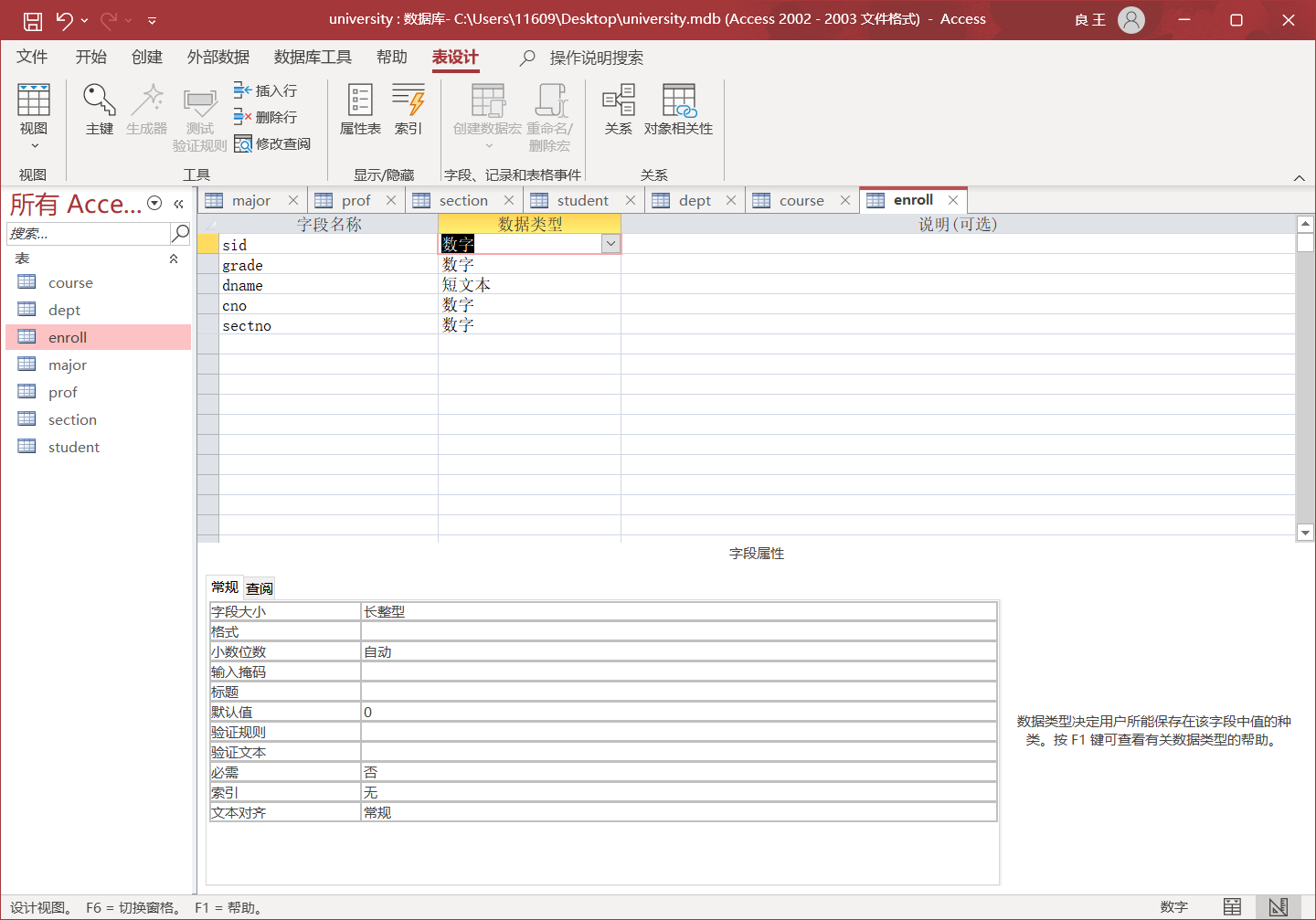
In Access:

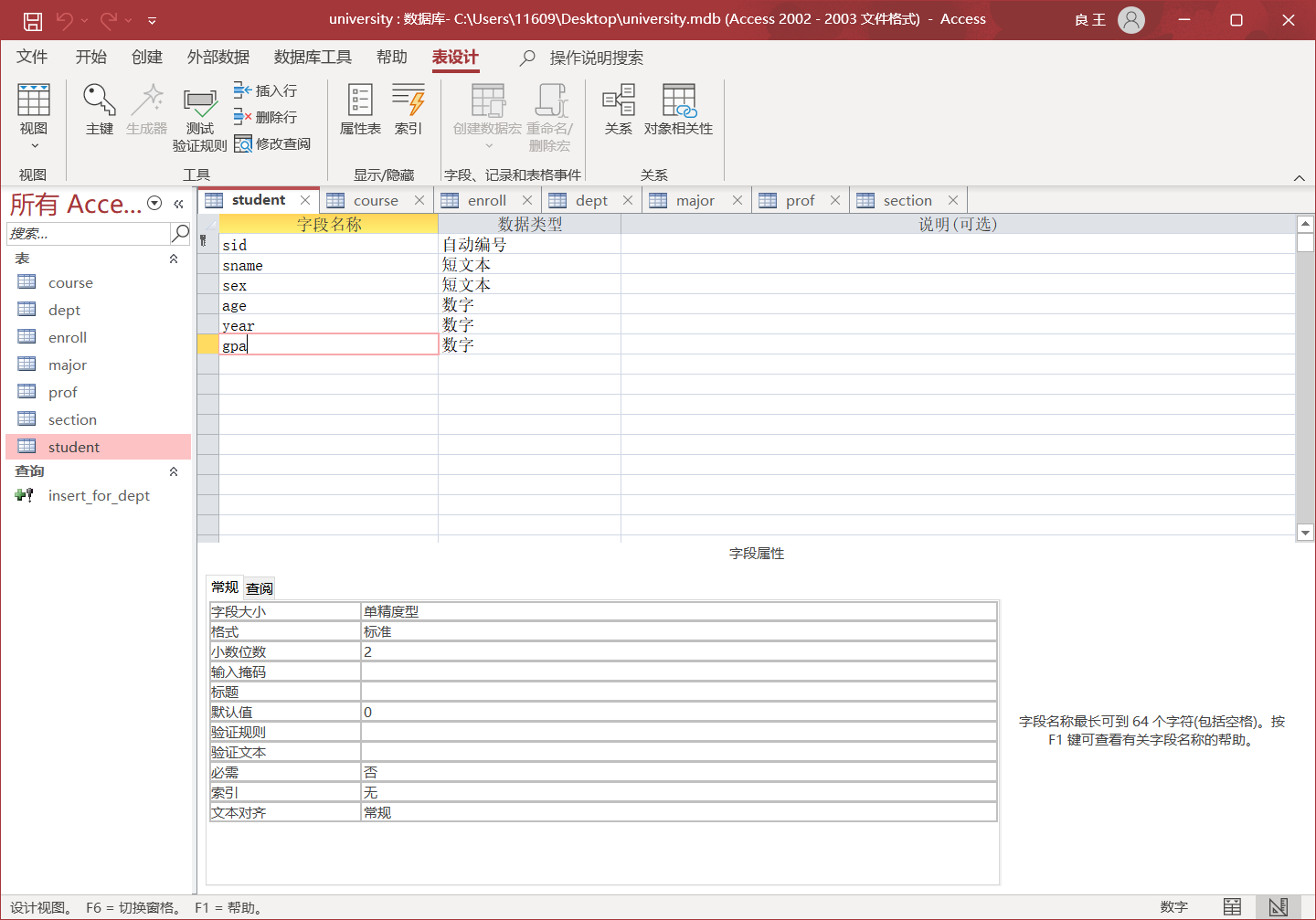


Now, before the data injection. Let’s focus on some noteworthy details.

In Access, every first field would be considered as primary key in spite of data type. (You may examine it in design view) I should only demonstrate for once that how I eliminated that unwanted behavior (except for “student” table). Select the field and click the “primary key” button to cancel primary key effect. Adjust “gpa” to float type and its demonstration form to be “standard”, which can display exactly what we expect like “0.20” (from data with sid of 80).



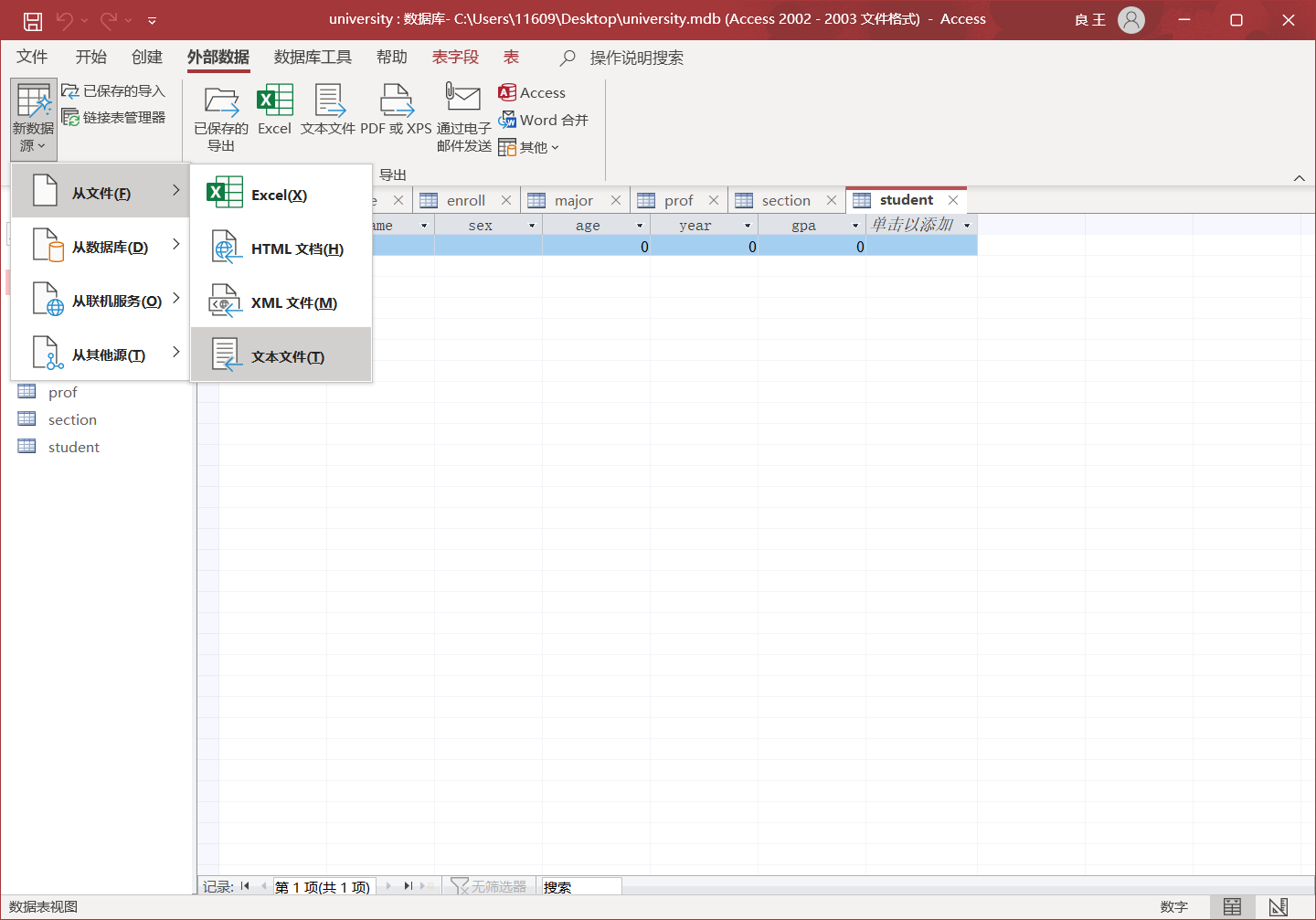


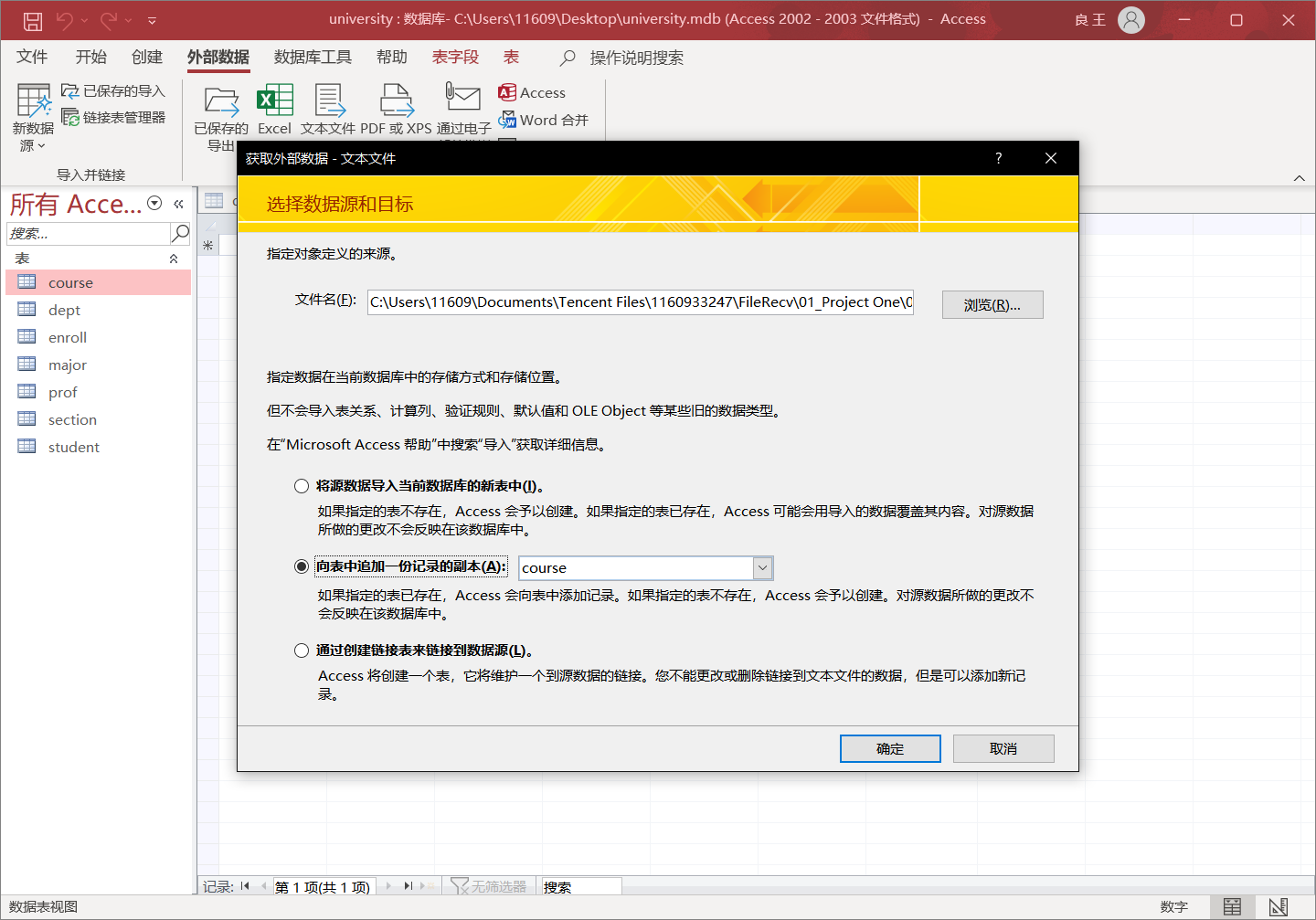


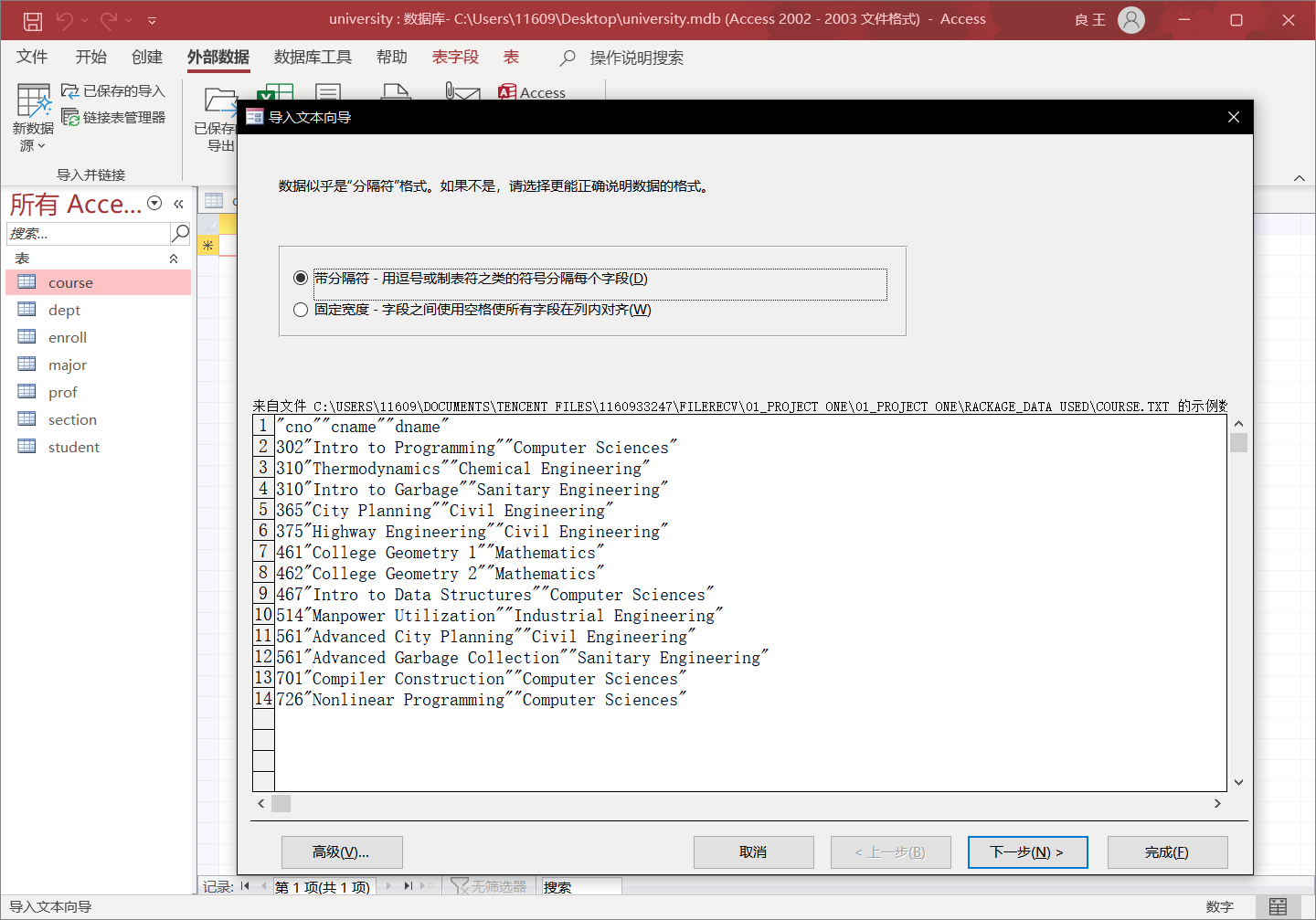
# **Second: data injection & first sql**

This part is literally a simple and dull one. You may get through this part swiftly since there’s nothing in particular worth mentioning.

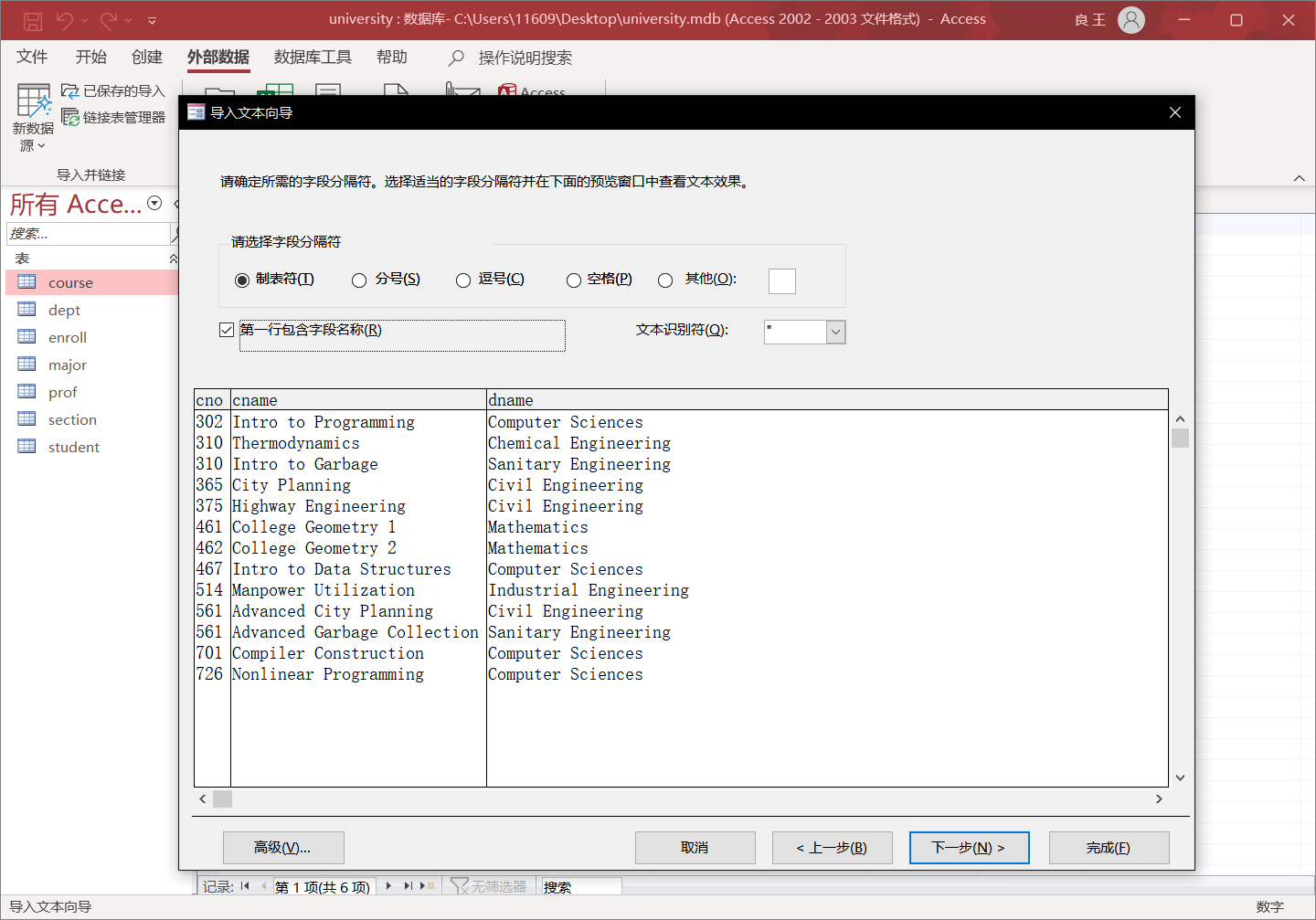
One instance for data importing.

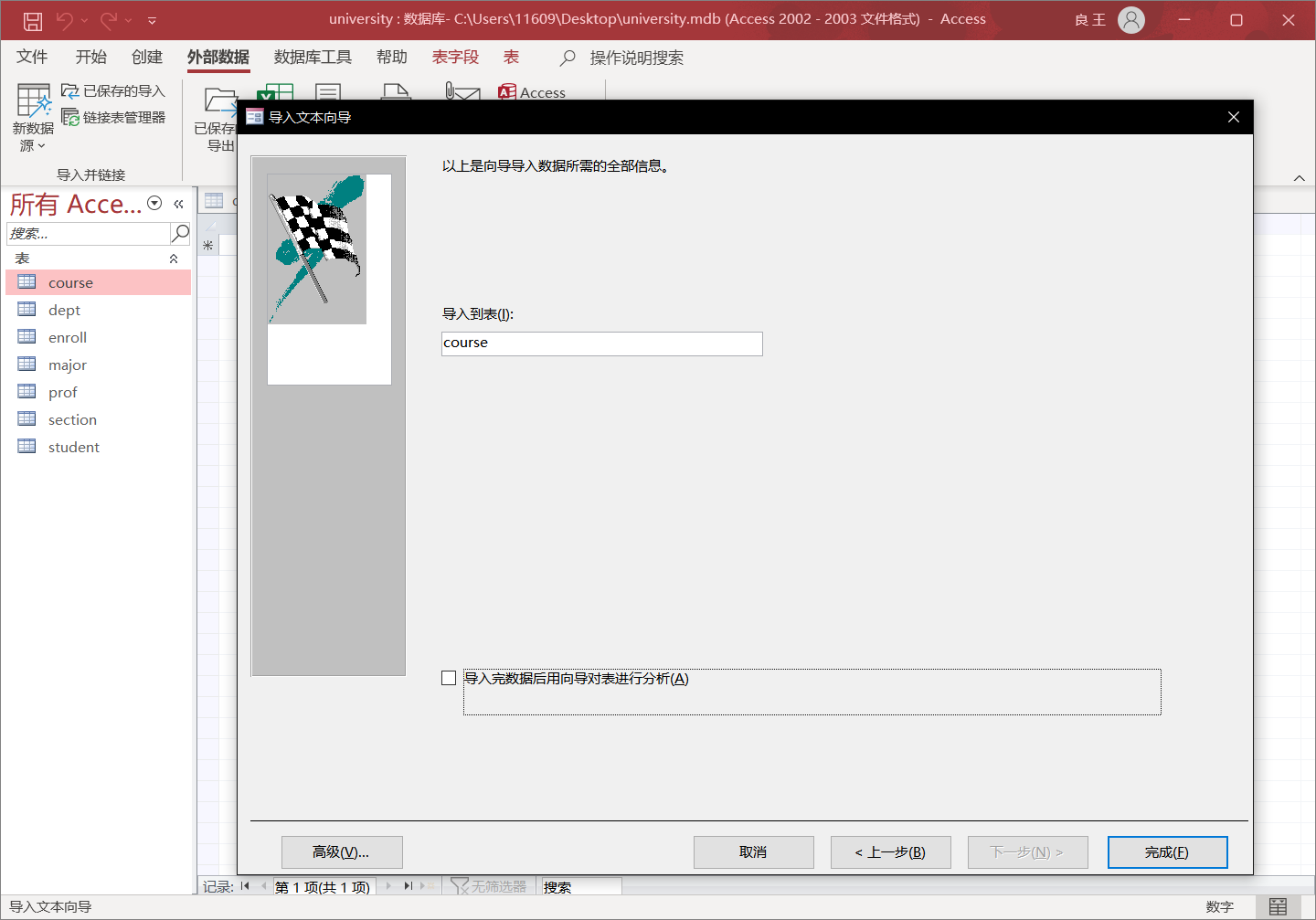




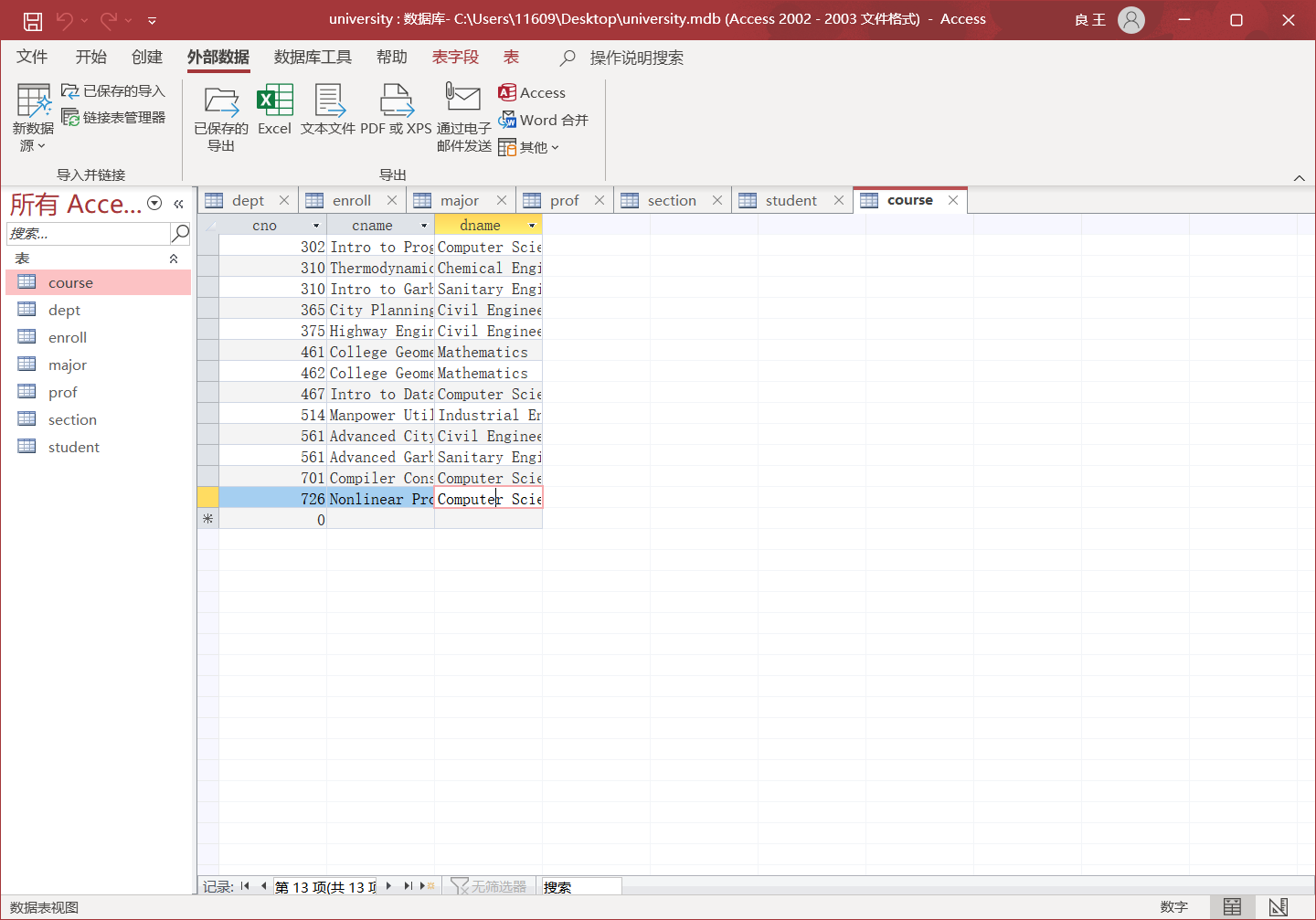


The only trap should be this unnoticeable choice.

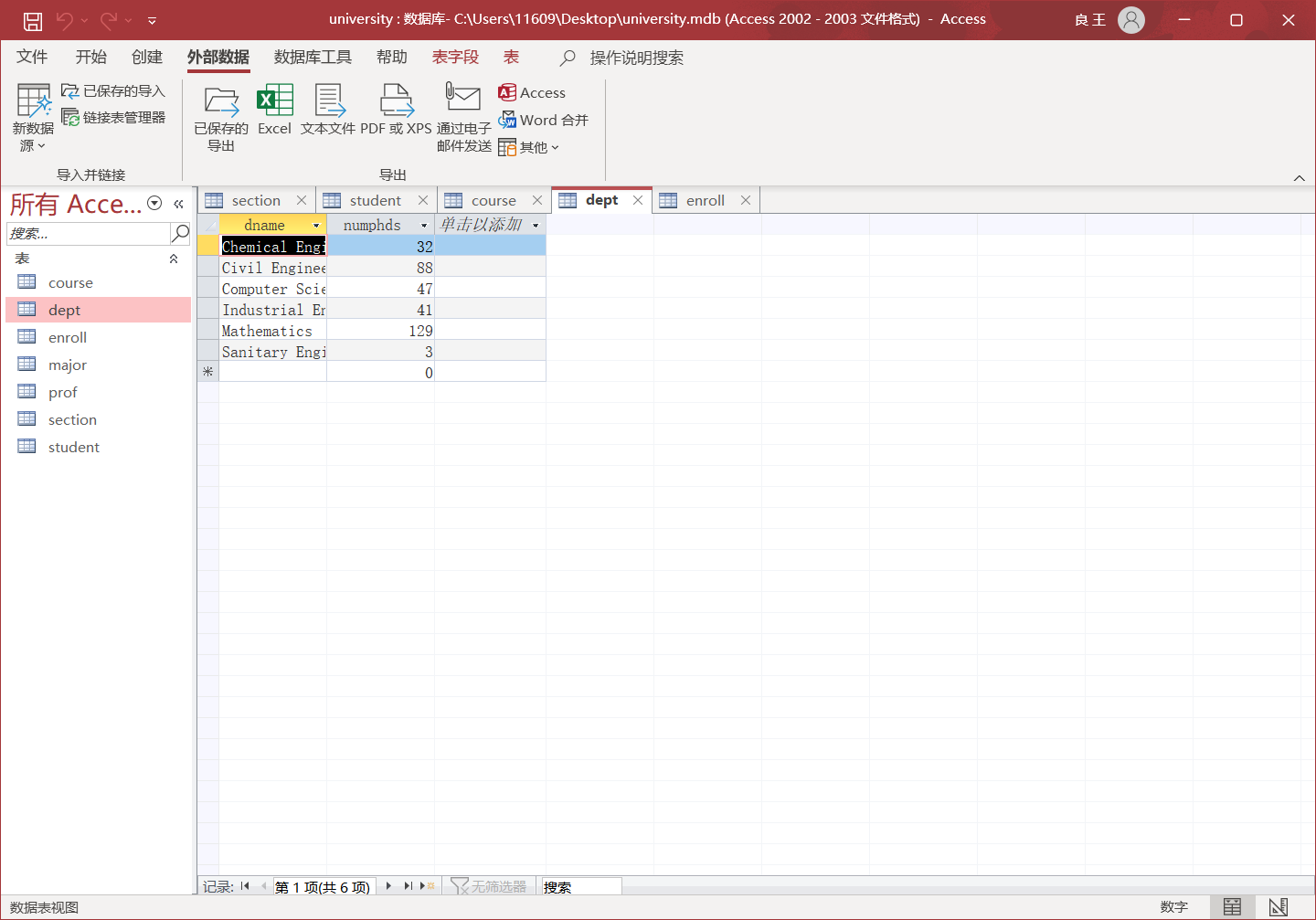


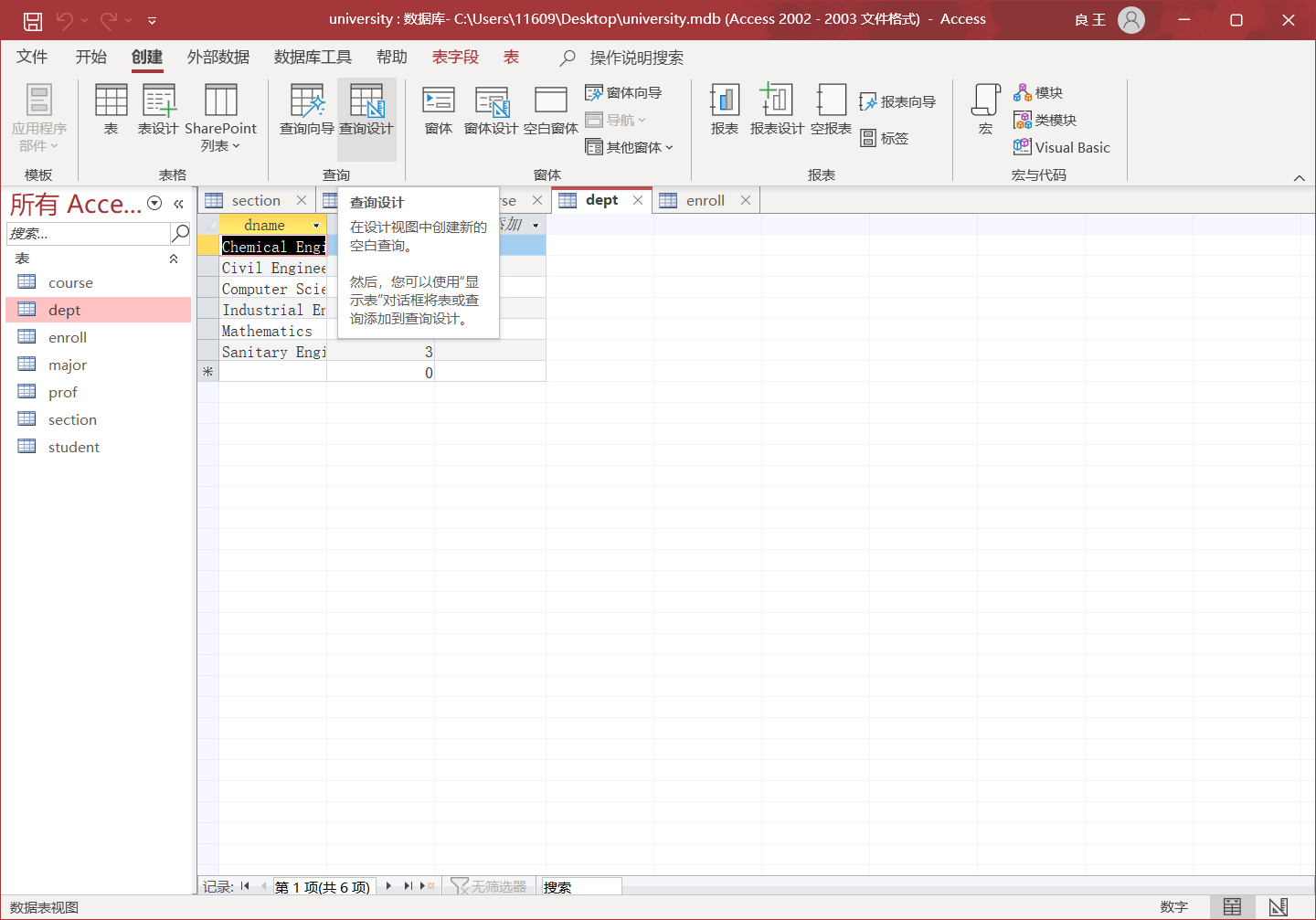


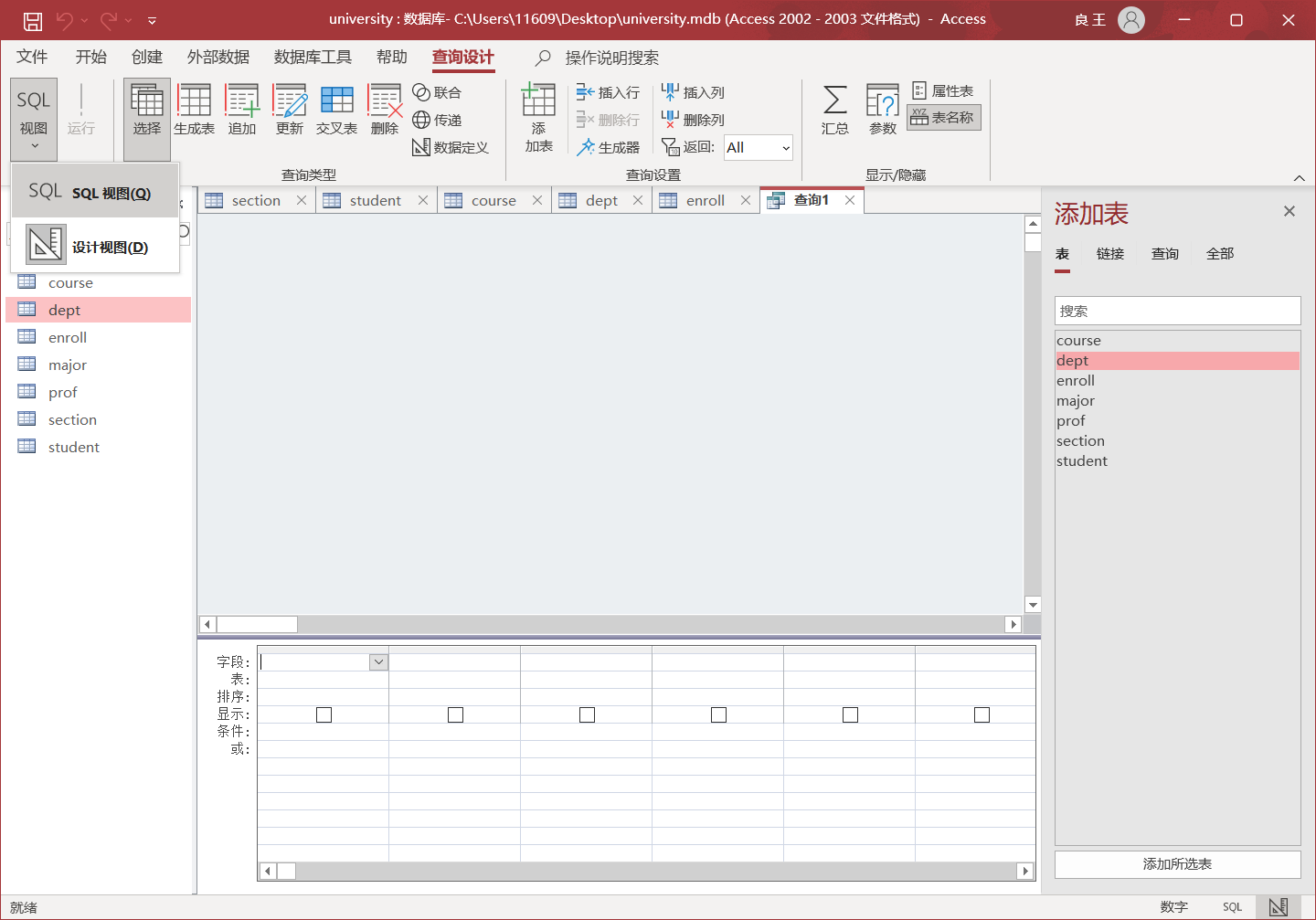
No errors reported, just repeat these steps.

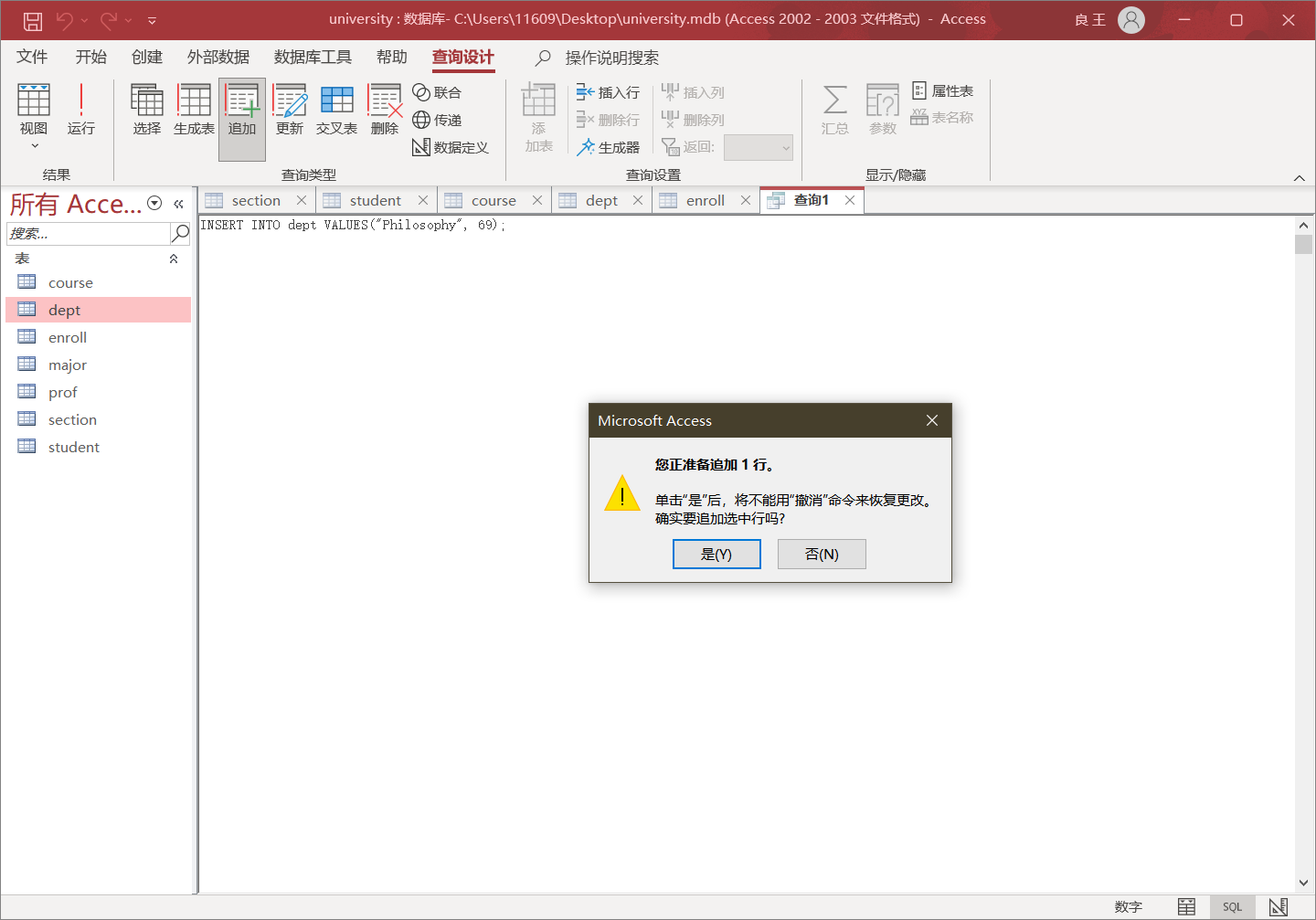


All injections well done, we are eligible to utilize sql to insert a new recode, here “dept” table as one instance.

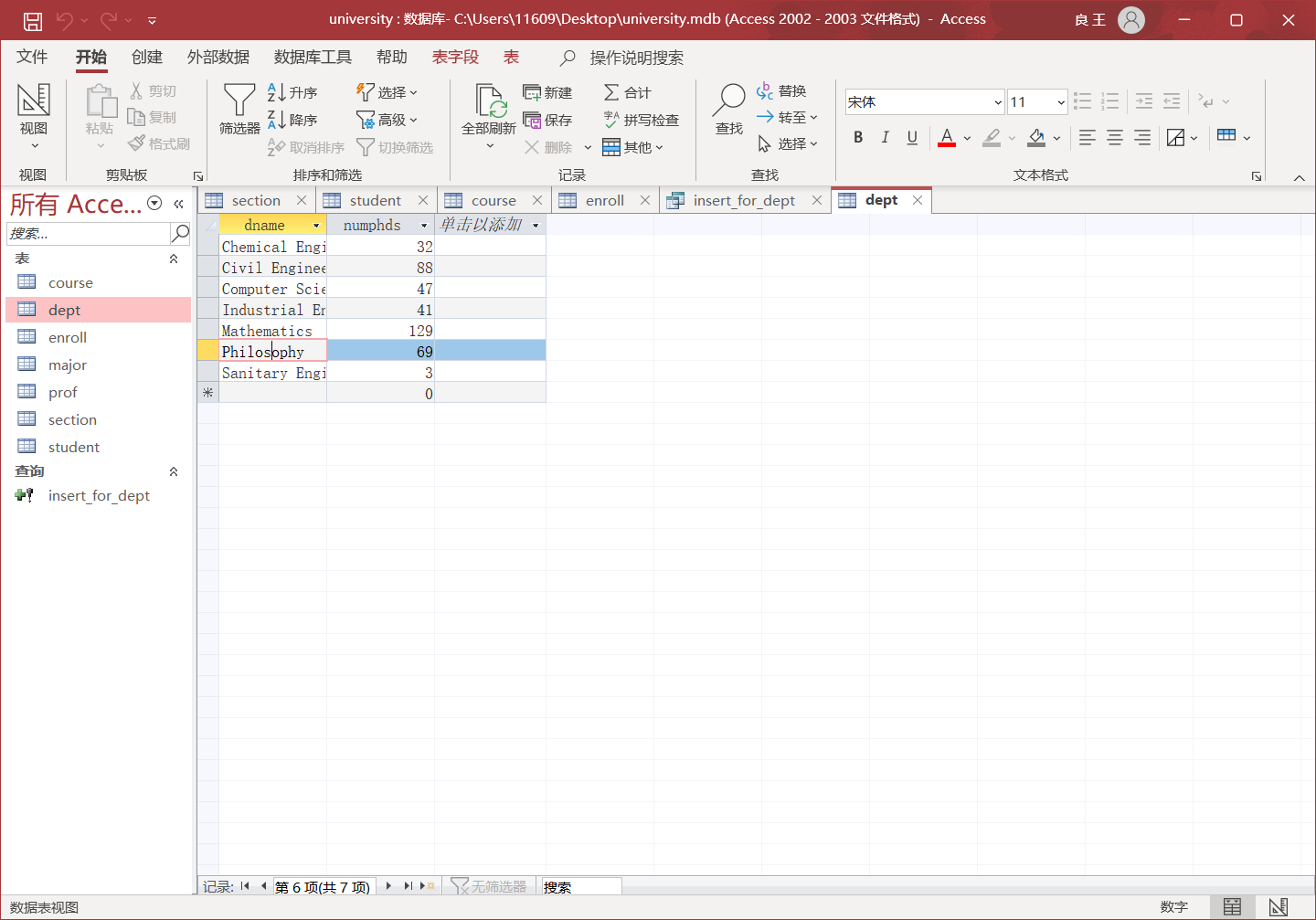








Save the query, re-access the “dept” table. See, a new record is inserted here.



# **Third: “lorem ipsum” & sql en masse**

It is this part that everything becomes a little bit tricky (thanks to the weird database choice of Access :( ). We are now prepared to take off the gloves and start to task us with some programming. (The programs codes are not provided in zip, you may contact me if required).

Data generation: Utilizing the module of faker.js, it is quite a easy task to work through. I prepared the “.ts” standing for typescript file as my choice. ~~Just another chance to get myself comfortable with ts.~~ I am positive that code is semantically clear so it can be examined with no burden. Or just see “project1.csv” directly. If you were to run the code on your own machine, remember to use npm install (node.js environment required) and npm install ts-node to help you run ts file directly.

*import* { *faker* } *from* '*./node\_modules/@faker-js/faker*'

*import* *\** *as* fs *from* '*fs*'

*type* gender = '*female*' | '*male*'

*type* sex = '*f*' | '*m*'

*type* year = 1 | 2 | 3 | 4 | 5

*class* Student {

*sid:* *number*

*sname:* *string*

*sex:* *sex*

*age:* *number*

*year:* *year*

*gpa:* *number*

*constructor*(

*sid: number*,

*sname: string*,

*sex: sex*,

*age: number*,

*year: year*,

*gpa: number*

) {

*this*.sid = sid

*this*.sname = sname

*this*.sex = sex

*this*.age = age

*this*.year = year

*this*.gpa = gpa

  }

*toString*() {

*return* Object.*values*(*this*)

  }

}

*let* data*:* *Student*[] = []

*function* *getName*(*gender: gender*)*:* *string* {

*let* isGivenName*:* *boolean* = faker.datatype.*boolean*()

*let* isMiddleName*:* *boolean* = faker.datatype.*boolean*()

*let* givenName*:* *string* = isGivenName ? faker.name.*firstName*(gender) : ''

*let* middleName*:* *string* = isMiddleName

    ? faker.name.*middleName*(gender)[0] + '*.*'

    : ''

  givenName = givenName ? '' + givenName : ''

  middleName = middleName ? '' + middleName : ''

*let* lastName*:* *string* = faker.name.*lastName*(gender)

  lastName = givenName || middleName ? lastName + '*,*' : lastName

*return* lastName + givenName + middleName

}

*function* *getSex*()*:* *gender* {

*return* faker.name.*sex*() *as* *gender*

}

*function* *getAge*()*:* *number* {

*return* faker.datatype.*number*({ *min*: 18, *max*: 50 })

}

*function* *getYear*()*:* *year* {

*return* faker.datatype.*number*({ *min*: 1, *max*: 5 }) *as* *year*

}

*function* *getGpa*()*:* *number* {

*return* faker.datatype.*number*({ *min*: 1, *max*: 4, *precision*: 0.01 })

}

*function* *studentsGenerator*()*:* *Student*[] {

*let* sid*:* *number*

*let* sname*:* *string*

*let* sex*:* *sex*

*let* age*:* *number*

*let* year*:* *year*

*let* gpa*:* *number*

*for* (*let* i = 105; i *<* 105 + 6000; i++) {

    sid = i

    sex = *getSex*() *as* *sex*

    sname = *getName*(sex *as* *gender*)

    sex = sex[0] *as* *sex*

    age = *getAge*()

    year = *getYear*()

    gpa = *getGpa*()

    data.*push*(*new* *Student*(sid, sname, sex, age, year, gpa))

  }

*return* data

}

data = *studentsGenerator*()

fs.*writeFile*(

  `${*\_\_dirname*}*/project1.csv*`,

  data

    .*map*(

      (*item*) *=>*

        `${*Object*.*values(item)*.*reduce(*(*prev*, *cur*, *index*) *=>* {

*if (index* === *1* || *index* === *2)* {

*return prev* +'*,"*'+ *cur* +'*"*'

}

*return prev* +'*,*'+ *cur*

}*)*}`

    )

    .*join*('*\n*'),

  (*err*) *=>* {

*if* (err) {

      console.*log*('*Process: write new failed*')

*return*

    }

    console.*log*('*Process: write new succeed*')

  }

)

What the “csv” data file looks like (6000 records):

105,*"Stark",*"f",*20,1,3.8*

106,*"West",*"f",*39,3,2.72*

107,*"Ratke, Samuel B.",*"m",*38,4,1.34*

108,*"MacGyver, L.",*"f",*46,5,2.67*

109,*"Runte, Mamie",*"f",*49,4,2.73*

110,*"Hane, J.",*"f",*46,1,1.78*

111,*"Kulas, Danielle",*"f",*19,3,3.74*

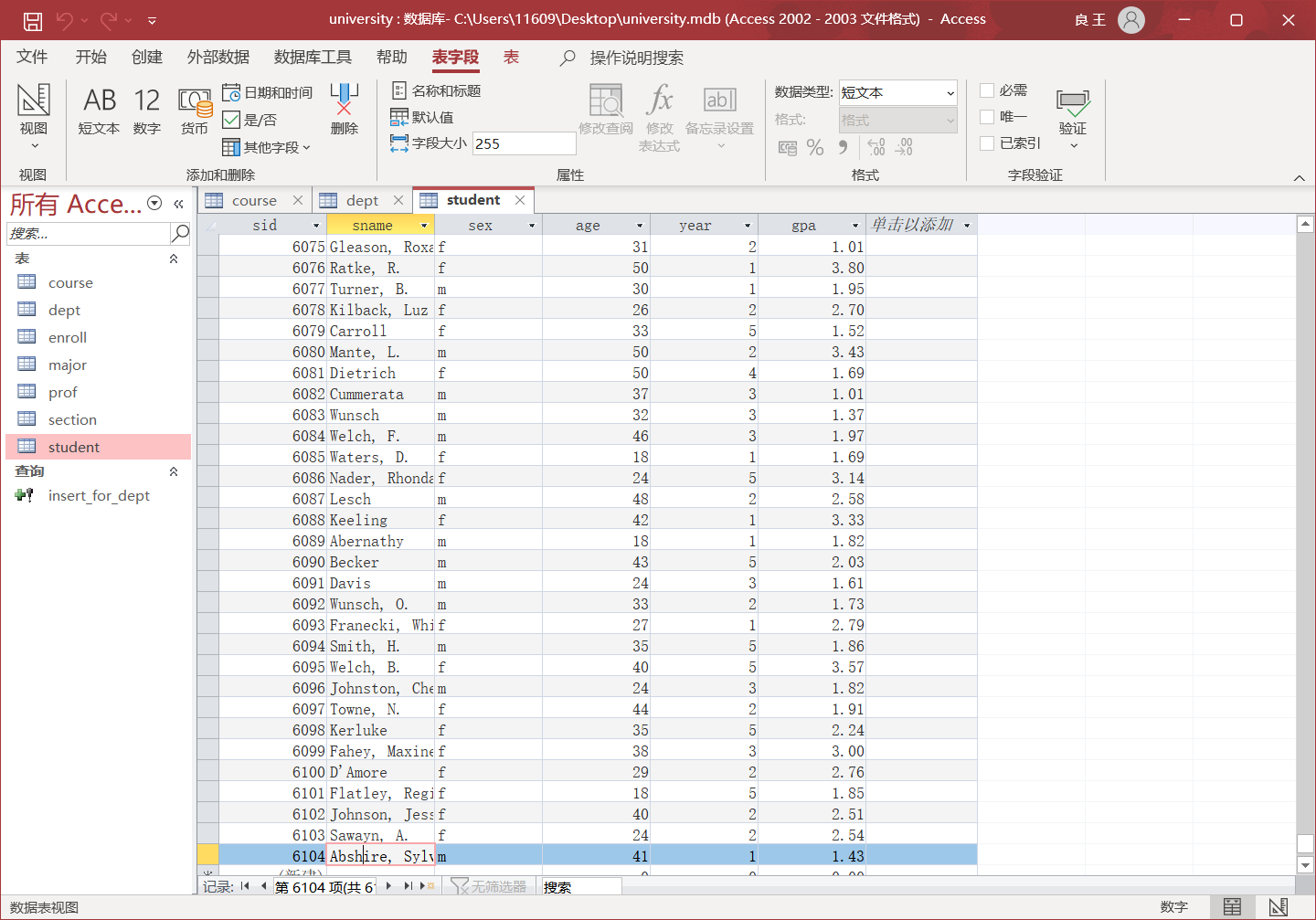
112,*"Crist, Hugh",*"m",*30,5,1.6*

113,*"Lehner, Melody S.",*"f",*35,2,3.76*

And for the substantial sqls, see key java code snippet (required maven to run the whole project):

for (int i = 0**;** i < recordsCount**;** i++) {  
 String sql = "INSERT INTO student (sid, sname, sex, age, year, gpa) VALUES (?, ?, ?, ?, ?, ?)"**;** PreparedStatement preparedStatement = connection.prepareStatement(sql)**;** preparedStatement.setInt(1**,** Integer.*parseInt*(dataList.get(i)[0]))**;** preparedStatement.setString(2**,** dataList.get(i)[1])**;** preparedStatement.setString(3**,** dataList.get(i)[2])**;** preparedStatement.setInt(4**,** Integer.*parseInt*(dataList.get(i)[3]))**;** preparedStatement.setInt(5**,** Integer.*parseInt*(dataList.get(i)[4]))**;** preparedStatement.setFloat(6**,** Float.*parseFloat*(dataList.get(i)[5]))**;** rowAffected += preparedStatement.executeUpdate()**;**}

Here’s where we end. (original 104 + my 6000)



# **Conclusion**

This project is basically getting oneself comfortable with the MS Access program and getting to know database structures. Since I have worked several times with mysql DBMS beforehand, I don’t find Access a charming one to work with(maybe the opposite). To conclude, I learned:

* What it means to be a **structure** query language.
* How data is organized in a relational database.
* Beware of the **data type**.
* ~~JDBC is such an amazing idea, one interface for all with same expectation.~~
* ~~Access is in literally rare use. No much supporting library.~~
* Speculation on next experiment be like: primary key foreign key separate tables.